A New Approach to Version Control

John Plaice and William W. Wadge

Abstract—**We present a new approach to the control of versions of software and other hierarchically structured entities. Any part of a system, from the smallest component to a complete system, may exist in different versions. The set of all possible versions under the refinement relation forms a partial order (in fact, a lattice). The fact that version V approximates version V' in this order means that V is *relevant* to V' in this sense: when constructing version V' of a system, we can sometimes use version V of a component if nothing more appropriate is available. More precisely, a particular version of an entire system is formed by combining the most relevant existing versions of the various components of the system. We call this the *variant structure principle;* it makes precise the idea that components of a given version of the system can be *inherited* by more refined versions of the system. We give an algebraic version language which allows histories (numbered series), subversions (or variants), and joins. In particular, the join operation is simply the lattice least upper bound. The join operation, together with the variant structure principle, provide a systematic framework for recombining divergent variants. We demonstrate the utility of this approach through LEMUR, a programming environment for modular C programs, which was developed using itself. Finally, we show how this notion of versions is related to the possible world semantics of intensional logic.**

Index Terms— **Modules for C, programming environments, programming languages, software engineering, version control.**

I. INTRODUCTION

SOFTWARE systems undergo constant evolution. Specifications change, improvements are made, bugs are fixed, and different versions are created to suit differing needs. As these changes are made, families of systems arise, all very similar, yet different. Handling such changes for a large system is a nontrivial task, as its different components will evolve differently.

Existing version control and software configurations systems have succeeded in solving some of the problems of dealing with this evolution. Pure version control systems such as SSC [22] and RCS [28], [30], using delta techniques to save storage space, keep track of the changes made by the different programmers to a file. Other space saving techniques have also been developed [8], [16], [18], [29]. Software configuration systems such as MAKE [7] allow for the automatic reconfiguration of a system when changes are made to a component. Also, more detailed analysis of changes to components reduces much useless compiling [24], [31].

Manuscript received April 15, 1991; revised July 6, 1992. Recommended by Marvin Zelkowitz.

J. Plaice is with the Département d'Informatique, Université Laval, Québec, Québec, Canada G1K 7P4.

W. Wadge is with the University of Victoria, Victoria, British Columbia, Canada V8W 3P6.

IEEE Log Number 9206897.

Integrated systems attempt to combine these ideas. Among the better known are System Modeller [11], [23], Tichy's work at CMU [26], [27], GANDALF [4], [9], [17], Adele [1], [2], [5], [6], DSEE [12], Jasmine [15], `shape` [13], [14], and Odin [3]. These systems, to a greater or lesser degree, allow for the development of large projects being developed by many different programmers. They use software databases, version control for the files, sometimes also for modules, as well as allowing the restriction of certain tasks to certain individuals. Some of them integrate versioning of files right into the operating system.

Despite these advances, the integration of hierarchically-structured entities and version control is still not satisfactory. Using a system such as RCS and SCCS, for each file, there is a tree of revisions. The trunk is considered to be the "main" version, and the branches correspond to "variants." Often, when a number of changes have been made to a variant, the changes are "merged" (sometimes textually) back into the trunk. The tree structure does not show how this merge took place.

If integrated environments, such as Adele, are used, then for each module family, there can be variants of the specification. For each specification, there can be variants of the implementation. And then for each implementation, there is an RCS-like structure for the development of the implementation.

In both cases, a tree structure is used for versions; yet, the tree structure is not appropriate for software development, because of the constant merging of different changes to the same system. A directed acyclic graph (dag) would be more appropriate. For example, suppose a program is written to work with a standard screen in English. Two people independently modify the program. The first adds a graphics interface, and the other changes the error messages to French. And then someone asks for a version that has both graphics and French messages. This new version inherits from its two ancestors, just as classes can inherit from several ancestors in object-oriented programming.

The concept of variant is not fully developed. Parnas [19] described the need for families of software and showed that having variants is a good idea, however the concept has still not been formalized. In the discussion on variants in [32], no one could give a definition of variant. In [14], we read, "We suspect that it is still an unsolved problem of software engineering to produce *portable software designs* in the sense of predicting and planning the possibility that certain modules of a system sprout variant branches. It is still a fact that variants *happen*."

Perhaps the problem is that variants must be planned, instead of being allowed to happen. Furthermore, one should be able to

refer to the version of a complete system, in the same language as one does for the versions of components.

This paper addresses the concept of variant, and how the variants of a complete system relate to the versions of individual components. Section II presents the need for versions of complete systems, and informally presents how versions of components and complete systems should interact. Section III formally presents an algebra for versions, which allows subversions and join versions, along with a refinement relation between versions; the version space therefore creates a lattice. Section IV formally presents the relationship between versions of complete systems and of components, using the variant substructure principle. Section V illustrates how this version language is used in an already existing C programming environment. Finally, Section VI discusses some of the ideas, presenting possible extensions, as well as showing that they could be integrated into existing configuration management systems.

II. Global Versions

The main weakness of existing tools is that the different versions of a component have only a local significance. It might be the case, for example, that there is a `third` version of component A and also a `third` version of component B. But there is no *a priori* reason to expect any relationship between the `third` versions of separate components.

The only exception is in the concept of variant. For integrated environments such as Adele, a variant represents a different interface to a module, and so has more than local significance. But the components of the implementation of each interface are completely separate, thereby creating a situation of code duplication, or of juggling with software configuration.

This lack of correspondence between versions of different components makes it difficult to build a complete system automatically. Instead, users are allowed to mix and match different versions of different components arbitrarily. These tools give the users the "freedom" of building any desired combination, but they also burden them with the responsibility of deciding which of the huge number of possible combinations will yield a consistent, working instance of the system.

In our approach, however, version labels (which are not necessarily numbers) are intended to have a global, uniform significance. Thus the `fast` version of component A is meant to be combined with the `fast` version of component B. Programmers are expected to ensure that these corresponding versions are compatible.

One advantage of this approach is that it is now possible to talk of versions of the *complete system*—formed, in the simplest case, by uniformly choosing corresponding versions of the components. Suppose, for example, that we have created a `fast` version of every component of a (say) compiler. Then we build the `fast` version of the compiler by combining the `fast` versions of all the components.

Of course, in general it is unrealistic to require a distinct `fast` version of every component. It may be possible to speed up the compiler by altering only a few components, and only these components will have `fast` versions. So we extend our configuration rule as follows: to build the `fast` compiler we take the `fast` version of each component, if it exists; otherwise we take the ordinary "vanilla" version.

We generalize this approach by defining a partially ordered algebra of version labels. The partial order is the refinement relation: $V \sqsubseteq W$, read as "V is refined by W," or "V is relevant to W," means (informally) that W is the result of further developing version V. The basic principle is that in configuring version W of a system, we can use version V of a particular component if the component does not exist in a more relevant version. That is, we can use version V of the component as long as the component does not exist in version V' with $V \sqsubseteq V' \sqsubseteq W$.

We then use this refinement ordering to automate the building of complete system. The user specifies only which version of the complete system is desired; our variant structure principle defines this to be the result of combining the most relevant version of each component.

III. Version Space

In this section, we introduce our version algebra, giving the rules and practical applications of each of the version operators. The simplest possible algebra would allow only one version. We call this version the *vanilla* version, written ϵ, the empty string.

A. Project History

The simplest versions are those that correspond to successive stages in the development process: version 1, version 2, version 3, etc. An obvious extension is to allow subsequences: `1.1`, `1.2`, or `2.3.1`.

Having such a version control system would not just facilitate maintenance. It would also aid the recovery from error, be it physical, such as the accidental destruction of a file, or logical, such as the introduction of a flawed algorithm. Furthermore, it would allow the recuperation of previously rejected ideas. It is not uncommon for an idea to be conceived, partially thought through and rejected, only to be needed six months later.

It was to solve this kind of problem that programs such as SCCS and RCS were designed; in fact, the notion of numeric string to keep track of the successive stages is quite suitable. However, the . of RCS has two different meanings. Version `1.2.3.4` actually means subversion `3.4` of version `1.2`, and is not on the trunk of the version tree. Versions `1.2.3.4` and `1.3` are therefore incomparable, even though it would appear from the figures that `1.3` succeeds `1.2.3.4`.

In our version space, numeric versions can only build one branch. Subversions must be used to create forks. Our initial set of possible versions can be described by the following grammar:

$$V ::= \epsilon \mid N$$
$$N ::= n \mid N.n$$

where n is a nonnegative integer.

The refinement order as described earlier indicates how one version is derived from others. This order, written $\sqsubseteq$, must be well founded and transitive:

$$\epsilon \sqsubseteq V$$
$$\frac{V \sqsubseteq V' \quad V' \sqsubseteq V''}{V \sqsubseteq V''}.$$

For our current set of versions, we use the intuitive, dictionary order:

$$\frac{n \leq m}{n \sqsubseteq m}$$
$$N \sqsubseteq N.n$$
$$\frac{N \sqsubseteq M \quad N \text{ is not a prefix of } M}{N.n \sqsubseteq M}.$$

So, for example, $1.2.3.4 \sqsubseteq 1.3 \sqsubseteq 2.4.5$.

How numeric versions would be used would depend on the environment in which they are used. One example would be to keep a complete record of all changes to all files. This could be done by having six-number version names, corresponding to the date, as in `1992.06.18.11.18.29`. Another approach would be to use more numbers as editing of a file is taking place, and fewer numbers for the "real" versions, that have some meaning: versions `1.2.1`, `1.2.2`, and `1.2.3` could then correspond to the successive edits of version `1.2`, ultimately yielding version `1.3`.

B. Differing Requirements

If a piece of software is going to be used by people in differing environments, it is likely that the requirements of those users will differ. One of the most important differences would be at the level of user interface. Some aspects are a matter of personal taste, such as does one prefer to use graphics and menus, or does one prefer text? Others are a necessity. A Syrian would want to read and write in Arabic, a Japanese in katakana, hiragana, and kanji, and a Canadian would want to be able to choose between English and French. Even if the essential functionality were the same, the differences in user interface would be significant.

But differences in functionality can also appear. For example, most Lisp systems are Brobdingnagian,[1] as everything, *including* the kitchen sink, is included. Yet the typical Lisp user has no need for many of the packages that are offered. Rather than being forced to take the mini or the maxi version, users should be able to pick and choose among the packages that they need. For this particular example, autoload features can be used, but this is not the case for all systems.

Differences in implementation may also arise as one ports a system from one machine to another. The versions for machines X and Y may be identical, but differ with that for machine Z.

To handle these problems, we need to introduce the concept of a *subversion,* called *variant* in many systems. This problem was partially addressed in SCCS and RCS, with the introduction of branches; unfortunately, relying on the numeric strings to identify the branches becomes very unwieldy. We choose the path of *naming* the subversions. Our new space of possible versions becomes

$$V ::= \epsilon \mid N \mid x \mid V\%V$$
$$N ::= n \mid N.n$$

where x is any alphanumeric string. For example, the `graphics%mouse` version of a user interface would be the mouse subversion of the graphics version of the user interface. Parentheses can be inserted at will to reduce ambiguity.

Unlike in RCS, names are not variables, but constants. They do not represent anything except themselves. Under the refinement relation, they are all incomparable.

We need one more axiom for subversions:

$$V \sqsubseteq V\%V'.$$

We consider the % operator to have ϵ as identity and to be associative:

$$\epsilon\% \equiv V$$
$$V\%\epsilon \equiv V$$
$$(V\%V')\%V'' \equiv V\%(V'\%V'').$$

Subversions can be very powerful. For example, consider the task of simultaneously maintaining separate releases. This is a common example, as it is normal to have a working version and a current version being developed, yet it is difficult to handle properly. Suppose that the two current releases are `2.3.4` and `3.5.6`. If we wish to make repairs to `2.3.4`, a subversion is required. For if we were to create a version `2.3.4.1` to fix the bug, then that version would still be considered to be anterior to `3.5.6`, which does not correspond to reality. Instead, we would want a `2.3.4%bugfix`.

The reader might wonder why the grammar allowed for $V\%V$ rather than $V\%x$. Consider the task of Maria and Keir each working separately on their own subsystems, each with their own sets of versions and subversions. When their work is merged, to prevent any ambiguity, all of Maria's versions could be preceded by `Maria%`; similarly for Keir.

C. Joins of Versions

Subversions allow for different functionalities. It is not uncommon, however, for different subversions to be compatible. For example we can easily imagine wanting a Japanese Lisp system with infinite precision arithmetic with graphics for machine X. To handle this sort of thing, we need to be able to *join* versions. Our Lisp version would be `Japanese+graphics+infinite+X`. Our final space of versions becomes:

$$V ::= \epsilon \mid N \mid x \mid V\%V \mid V + V$$
$$N ::= n \mid N.n.$$

To make the order complete, we add two more axioms:

$$V \sqsubseteq V + V'$$
$$\frac{V_1 \sqsubseteq V_1' \quad V_2 \sqsubseteq V_2'}{V_1 + V_2 \sqsubseteq V_1' + V_2'}.$$

[1] Brobdingnag was an imaginary country of giants in Jonathan Swift's *Gulliver's Travels.*

The + operator is idempotent, commutative, and associative, and left distributes %:

$$V + V \equiv V$$
$$V + V' \equiv V' + V$$
$$(V + V') + V'' \equiv V + (V' + V'')$$
$$V\%V' + V\%V'' \equiv V\%(V' + V'').$$

The + operator is defined so that it is the least upper bound operator induced by the $\sqsubseteq$ relation. Consider versions V_1 and V_2. Version $V_1 + V_2$ is the least upper bound of V_1 and V_2 if and only if for all V such that $V_1 \sqsubseteq V$ and $V_2 \sqsubseteq V$, relation $V_1 + V_2 \sqsubseteq V$ also holds. Consider such a V: then $(V_1 + V_2) \sqsubseteq (V + V) \equiv V$. So $V_1 + V_2$ *is* the least upper bound.

The variant substructure principle, presented in Section IV, calls for the use of the most relevant components when a particular configuration is being built. If the + operator were not defined as the least upper bound operator, then the term "most relevant" would have no meaning.

The fact that the version language allows the join of independent versions does not necessarily mean that any arbitrary join actually makes sense. However, should a merge be made, and it does make sense, then the join perfectly addresses the need to describe the merge. Do note that no merging of text or code, as in [10], is taking place here. The merging only takes place at the version name, and the configuration manager must ensure that the components do make sense together. The only checking that takes place is syntactic, at the level of the version names (see Section IV).

D. Canonical Form

The equality axioms allow a canonical form for all version expressions. In fact, except for the commutative and associative rules of +, the equations simply become rewrite rules:

$$\epsilon\%V \rightarrow V$$
$$V\%\epsilon \rightarrow V$$
$$(V\%V')\%V'' \rightarrow V\%(V'\%V'')$$
$$V\%V' + V\%V'' \rightarrow V\%(V' + V'')$$
$$(V + V') + V'' \rightarrow V + (V' + V'')$$
$$\frac{V \sqsubseteq V'}{V + V' \rightarrow V'}.$$

For joins of versions, we must introduce a total order on subversions, which corresponds to some form of dictionary order, noted $\ll$:

$$\frac{N \ll x}{x \text{ alphabetically precedes } y \over x \ll y}$$
$$\frac{V \ll V'}{V \ll V'\%V''}$$
$$\frac{V \ll V''}{V\%V' \ll V''}.$$

With the dictionary order, we can get a canonical form for the joins as well:

$$\frac{V \ll V'}{V' + V \rightarrow V + V'}$$
$$\frac{V \ll V'}{V' + (V + V'') \rightarrow V + (V' + V'')}.$$

A few examples are given at the bottom of this page. It is assumed that + is right associative.

IV. Versions and Structure

Up to now, we have been referring indiscriminately to versions of complete systems and to versions of components. How do these interact?

As we already explained, we do not require that every component exist in every version. Instead, we consider the absence of a particular version as meaning that a more "generic" version is adequate; in the simplest case, as meaning that the vanilla version is appropriate. This means, for example, that when we configure the French version we use the French version of each component, if it exists; otherwise we use the standard one.

In general, however, the vanilla version is not always the best alternative. Suppose, for example, that we need the `Keir%apple%fast` version (which can be understood as the "fast version of Keir's apple version"). If a certain component is not available in exactly this version, we would hardly be justified in assuming that the vanilla one is appropriate. If there is a `Keir%apple` version, we should certainly use it; and failing that, the `Keir` version, if it exists. The plain one is indicated only if none of these other more specific versions is available.

Our general rule is that when constructing version V of a system, we choose the version of each component which most closely approximates V (according to the ordering on versions introduced earlier). We could call this the most relevant version. More precisely, to select the appropriate version of a component C, let $\mathcal{V}$ be the set of versions in which C is available. The set of relevant versions is $\{V' \in \mathcal{V} | V' \sqsubseteq V\}$.

```
x%y%z + x%a%b → x%(a%b + y%z)
(x + 4.1) + a%b → 4.1 + a%b + x
last + first → first + last
(x%y%z + a%(b%c + b%d%e)) + x%r → a%b%(c + d%e) + x%(r + y%z).
```

The most relevant version is the maximum element of this set—if there is one. If there is no maximum element, there is an error condition—and there is no version V of the given system.

We can generalize the principle as follows: suppose that an object S has components $C_1, C_2, \cdots, C_n$. Then the version of S that is most relevant to V is formed by joining versions V_1 of C_1, V_2 of $C_2 \cdots$ where in each case V_i is the version of C_i most relevant to V. Furthermore, the version of V constructed is $V_1 + V_2 + \cdots + V_n$.

This principle, which we call the variant structure principle, describes exactly the way in which subversions of a system can "inherit" components from a supervision. It also accords well with motivations given for the various version forming operators described in an earlier section. For example, it specifies that in constructing version `3.2` of an object, we take version `2.8.2` of an object which exists in versions `3.4`, `2.8.3`, `2.7.9`, `1.8`, `1.5.6`, and ϵ. It specifies that in building version `Keir%apple%fast` we select the `Keir%apple` version of a component that exists in versions `Keir`, `Keir%apple`, `Keir%fast`, `apple%fast`, `Maria%apple`, `fast`, and ϵ.

Finally, the principle also explains how + solves the problem of combining versions; for example, combining Maria's `orange` and Keir's `apple` versions (see Section III). The desired version for the system would be `Maria%orange+Keir%apple`. According to our rule, for each component we select the version most relevant to `Maria%orange` when no `Keir` version is available, and the version most relevant to `Keir%apple` when no `Maria` version is available. Thus, if the available versions of component `C` are `Keir%pear`, `Fred%apple`, `Keir`, `apple`, and ϵ, we take version `Keir`. On the other hand, if the versions available are `Keir%apple`, `Maria`, and ϵ, then there is no best choice and the system does not exist in the desired version.

Notice that it is possible to construct version `Maria%orange+Keir%apple` even when no component exists in that version. However, it also makes sense for an individual component to exist in a version with a + in it. This allows otherwise incompatible projects to be merged. Consider, for example, the situation just described; both Keir and Maria have seen fit to alter component `C`, which exists in both `Keir%apple` and `Maria` versions. According to our principle, we cannot form a `Maria%orange+Keir%apple` version of the system because there is no appropriate version of the component in question. The solution is for Keir and Maria to get together and produce a mutually acceptable compromise version that is compatible with both the `Keir%apple` and `Maria` variants of the system. If they can do so, they label this compromise component as the `Maria+Keir%apple` version of the component. Having done this, our principle now says that there is a `Maria%orange+Keir%apple` version of the whole system, because now the compromise version is the most relevant. (Recall that in the version ordering, both `Maria` and `Keir%apple` lie below `Maria+Keir%apple`, which in turn lies below `Maria%orange+Keir%apple`.)

V. LEMUR

To test our notion of versions, we added it to SLOTH, an existing software engineering environment for C programs developed by the authors. The resulting, evolved program is LEMUR. For a more complete presentation of SLOTH, as well as a comparison with related work, see [20].

A. SLOTH

SLOTH is a set of tools designed to facilitate the reusability of C programs. A system of modules was devised, more sophisticated than the method traditionally used for C programs. Each module is a UNIX directory: there are two interface files (`extern.i` for externally visible variables and `define.i` for manifest constants), two implementation files (`var.i` for local variables and `proc.i` for local routines), as well as the `body.i` containing the initialization code. The import file states which modules are needed for this module to run correctly.

SLOTH has three commands. The VM command is used to view files and the MM to modify them. The LKM command, original to SLOTH, builds, for each module, a `uselist` file containing a list of all the modules that it depends on by computing the transitive closure of the import dependencies. It then builds a `prog.c` file from all of the component files and compiles it; the resulting `prog.o` file is linked with the `prog.o` files of the other modules to make a complete system.

SLOTH has shown itself to be remarkably useful, and the intended goal of reusability is being met. The POPSHOP, in which several compilers are written, consists of over 100 different modules, and builds more than 10 different applications. The reader is asked to refer to [20] for more details.

A. LEMUR: SLOTH with Versions

LEMUR is an evolved form of SLOTH. LEMUR allows the user to create and label different versions of the individual files that make up a module. A label can be any element of the version space just described, that represented in a simple linear syntax and used as an extension of the file name. For example, if Keir needs a separate version of the procedure definition file of a module, he would create (inside the module) a new file `proc_Keir%apple.i`. And if his `apple` version had its own `fast` version, and if this `fast` subversion required further changes to the module's procedure definitions, he would create an additional file `proc_Keir%apple%fast.i`. Note the new files do not *replace* the old ones; the different versions coexist. Note also that not every file exists in every version. For example, the `fast` subversion of `Keir%apple` may require only a few changes. As a result, there will only be a few files with the full `Keir%apple%fast` label.

With LEMUR, only the basic component files have explicit, user-maintained versions. The users do not directly create separate versions of whole modules, or of applications. Instead, LEMUR uses the principle of the previous section to create, automatically, any desired version of an application.

Suppose, for example, that Maria would like to compile and run her `Maria%orange` version of the project (call it comp). She invokes the LEMUR configure command with comp as

its argument but with `Maria%orange` as the parameter of the `-v` option. LEMUR proceeds much as if the `-v` option were absent. It uses the import lists to form a "uselist" of all modules required; it checks that their `.o` files are up to date, recompiling if necessary; and then it links together an executable (which would normally be called `comp`). The difference, though, is that with each individual file it looks first for a `Maria%orange` version, instead of the vanilla one. And when the link is completed, the executable is named `comp_Maria%orange`.

If every file needed has a `Maria%orange` version, the procedure is straightforward. As we said earlier, however, we do not require that every file exist in the version requested. When the desired version is not available, LEMUR follows the principle of Section IV and selects the *most relevant* version available. In this instance, it means that if there is no `Maria%orange` version LEMUR looks for a `Maria` version; and if even that is unavailable, it settles for the vanilla version of the file in question. This form of inheritance, implicit in the variant structure principle, allows source code sharing between a version and its subversions.

LEMUR also follows the principle of Section IV when creating and labelling the `.o` files for individual modules. Suppose again that the `Maria%orange` version has been requested and that the relevant `.o` file of the `fred` module must be produced. LEMUR does this automatically, using in each case the most relevant versions of the internal `fred` files required and of the declarations imported from other modules. When the compilation is complete, LEMUR does not automatically label the resulting `.o` file `fred_Maria%orange.o`. It does so only if one of the files involved actually had the full `Maria%orange` label. Otherwise it labels the `.o` file as `fred_Maria.o`, assuming, at least that one `Maria` version was involved. And if all the files involved were in fact vanilla, the `.o` file produced is given the vanilla name `fred.o`. In general, it labels the `.o` file with the least upper bound of the versions of the files involved in producing it. The resulting label may be much more generic than the version requested; and this means that the same `.o` file can be used to build other versions of the system. The inheritance principle therefore allows us to share object code between a version and its subversions.

The high granularity of modules in SLOTH allows one to do all sorts of interesting things. For example, one could write a `test` version of a module interface that would allow a tester to look at the values of inner variables. The advantage is that the code itself (the implementation) would not change at all, nor would the files even be touched.

As the `import` file is separate, one can have one version of a module depend on one set of modules, and another version depend on another set of modules. In other words, the hierarchical structure (the shape) of the system can itself change from one version to another. In this sense, LEMUR actually goes beyond the principle of the previous section, which assumed the structure of a system to be invariant. However, we can easily reformulate the general principle by stipulating that every structured object has an explicit "subcomponent list" as one of its subcomponents. We then allow the subcomponent list to exist in various versions. When we configure the object, we first select the most relevant version of the component list; then we assemble the most relevant versions of the components appearing on this list. This is how LEMUR generalizes SLOTH's inport lists.

It is possible, using MAKE and RCS, to have versions of modules where different versions consist of completely different modules. If such is the case, different makefiles have to be written for each version, a lot of information has to be repeated. And a global makefile has to be written to ensure that the right version of the makefile is used to create the configuration. The whole process is quite complex.

With LEMUR, no makefiles need be written. Everything is done automatically. An extension to LEMUR, MARMOSET [21], allows different languages to be used, using a very simple configuration file, much simpler than standard makefiles.

C. Bootstrapping of LEMUR

To test our notion of version, LEMUR was bootstrapped: we used LEMUR to create versions of itself. The original SLOTH was written in a monolithic manner and did not handle versions. It was written, using the original version, into a modular form, much more suitable for maintenance and extension. Once this version (basic LEMUR, functionally the same as SLOTH) was working, it was used to create a system which allowed files to exist in multiple versions (true LEMUR). True LEMUR was then used to create subversion LEMUR, which allowed LEMUR to use not only basic versions, but also subversions, i.e., version `x.y`, `x.z`, and `x.y.a`.... A subversion of subversion LEMUR was created to accept numeric versions (numeric LEMUR). A new subversion was created to accept join versions (join LEMUR). Additional variants have also been created to allow different options; these were subsequently joined together.

D. Implementation

LKM builds modules one at a time, starting with those that do not depend on any other modules. It makes the most general version possible of each module. It then goes on to the more complex modules, still building the most general version possible; this version will, of course, depend on the versions of the modules that it depends on. Finally, it builds the most general possible version of the object file.

There can be situations where there is not a most general version. For example, one could ask for the `x+y` version, and for the one file there is an `x` version and a `y` version, but not a `x+y` version. For the VM and LKM programs, this is an error condition. On the other hand, MM asks the user if they wish to create a new file, and if so, what version of that file should be taken as the initial copy of the new version of the file.

VI. DISCUSSION

The problem of variants of software system is a difficult one. We claim that the language proposed in this paper is a step in the right direction. No difference is made between version, revision or variant. All subsystems are on the same level. If a variant evolves to the point where it becomes a completely

different product, that is just fine, and nothing special has to be done.

Of course, this paper in no way addresses how variants and versions are to be managed, in the sense of controlling how access to components of systems by programmers and users is made. The ideas in this paper do not put into question the need for software databases that restrict access to certain parts of a system so that it is not being modified in an uncontrolled manner.

A. The Language

Since we are using a lattice to describe our version space, one might ask if the meet of two versions has meaning. In fact, it does: `A%B` would be a version which is refined by each of `A,B` and `A+B`. For example, one could conceive of a common transliteration for Russian and Bulgarian, where, for example, if one writes *Dzhon* (John), the result would be Джон, this scheme being defined in `Russian%Bulgarian`. Then if one asked for the `Russian` version, then the `Russian%Bulgarian` file could be used if there were no `Russian` one.

With the current system, it is possible to have horrendously long version names. With the repeated use of + and % it might be difficult to figure out what is happening! One solution is to allow version variables; that is to introduce new versions defined in terms of existing ones. For example, suppose that the francophone Belgian users want a French-language mouse graphics version with infinite precision arithmetic. This corresponds to the version algebra expression `French+graphics%mouse+infinite`, which is clear enough but rather unwieldy. With version variables the user could introduce the definition

```
Belgian = French+graphics%mouse+infinite
```

and thereafter request the Belgian version or even use it in expressions like `Belgian+fast`.

In fact, the same effect can be achieved more elegantly by allowing *inequalities* rather than equalities. For example, the above definition can be given incrementally by the three inequalities

```
Belgian >= French
Belgian >= graphics%mouse
Belgian >= infinite.
```

Sometimes this complexity comes about because many modules are being named, and the versions within each module are different. In this case, the use of local version names, defined through inequalities as above, would allow the hiding of how the software was developed, one of the key goals of modules.

What we are proposing is a real version language, with constants and variables, with different scopes, defined using inequalities. Thus the next step would obviously be to add types. In fact, this is not surprising, since several configuration management systems allow for typed version names. The language component of a version name, for example, could be one of `Lucid` or `LUSTRE`.

B. Related Work

As we said in the Introduction, our approach to version control is original, so there is no work directly related. However, there is no reason that the ideas that were developed in this paper could not be applied to existing systems, and not just to software configuration systems (see following). In this sense, we will consider two systems which appear to be flexible enough for this change to be easily made: Odin and `shape`.

Odin [3] is a system that allows one to formalize the software configuration process. For each object, be it an atomic object or a tool that will manipulate the objects, axioms can be given declaring what the object does. One can then use pre- and postconditions to define the actions of tools. There is no reason that versioning cannot be added to the entire system. Atomic objects could have versions, and the rules defining what programs do would then pass the versions on. So, for example, the MM, VM, and IKM operations could then be defined in Odin.

`shape` [13], [14] is a system that attempts to integrate the better features of Adele, Make, and DSEE. There is an attributed file system interface, either to a standard file system or a database, which makes access to versions transparent to the user. The version language is in disjunctive normal form (OR's and AND's). If this version language were changed to allow +, and the variant substructure principle were applied, then `shape` would be generalized significantly.

C. Some Speculative Remarks

There is a close connection between the notion of version discussed here and the logical/philosophical notion of a "possible world" (see, for example, [25]). Possible worlds arise in a branch of logic, called "intensional logic," which deals with assertions and expressions whose meaning varies according to some implicit context. Usually the context involves space and time: the meaning of "the previous president" varies according to where the statement refers, e.g., the United States or France, and to when it refers (1989 or 1889). Other statements, e.g., "my brother's former employer," require more extensive information.

Obviously the notion of possible world, in its most literal form, raises mind-boggling philosophical questions. But, taken more formally, as indicating some sort of context (time, place, speaker, orientation), it has proved extremely useful in formalizing some hitherto mysterious and paradoxical aspects of natural language semantics (Montague being a pioneer in this area.

We can interpret an element of the version space as a possible world. In this possible world, there is an "instance" of the software in question; but this instance can differ from the instance in other possible worlds. For example, in this world, the error messages are in English, whereas in a neighboring world they are in French. The principle described earlier tells us how a compound object varies from one world to the next, provided we know how its parts vary. And LEMUR in a small way allows us to "visit" one of these worlds and construct

the instance of the software, without worrying about what the software looks like on the other worlds.

LEMUR is the result of intensionalizing one tool, e.g., SLOTH. We can surely imagine doing the same for other tools and even, if we are ambitious, UNIX itself. In this MONTAGUNIX we would specify, say with a command, which world we would like to visit—say, the `French+graphics%mouse` world. Having done that, it would give us the illusion that the appropriate instance is the only one that exists. In other words, when we examine the source, we would find only one copy; and only one copy of the `.o` files, and test files as well. These files could be scattered through a directory structure and we could move around.

Of course behind the scenes, MONTAGUNIX would be monitoring our activity and automatically choosing the most relevant version of every file we request. Other versions would be hidden from us. When we create a file, it would attach the appropriate version tag to it. MONTAGUNIX could give each developer the illusion of having their own private copy of a project in the same way that time sharing gave users the impression of having their own private computer. But the result is more sophisticated, because of the refinement relation between versions. However, we do not know what would be the implications when different users on the same network had conflicting software!

ACKNOWLEDGMENT

Many thanks to G. Brown who coded LEMUR. The quality of the code is exceptional, and we are pleased to announce that LEMUR is available from the first author.

REFERENCES

[1] N. Belkhatir and J. Estublier, "Experience with a database of programs," in *Ass. Comput. Mach. SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environments,* Palo Alto, CA, 1986; also in *SIGPLAN,* vol. 22, no. 1, pp. 84–91, Jan. 1987.

[2] ——, "Protection and cooperation in a software engineering environment," in *Advanced Programming Environments,* LNCS 244, Trondheim, Norway, June 1986, pp. 221–229.

[3] G. M. Clemm, "The Odin System: An object manager for extensible software environments," Ph.D. dissertation, University of Colorado, Boulder, 1986.

[4] L. W. Cooprider, "The representation of families of software systems," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1978.

[5] J. Estublier, "A configuration manager: The Adele data base of programs," in *Workshop on Software Engineering Environments for Programming-in-the-large,* Harwichport, MA, June 1985, pp. 140–147.

[6] J. Estublier and J.-M. Favre, "Structuring large versioned software products," in *30th Annual Int. Conf. Computer Software and Applications (COMPSAC'89),* Orlando, FL, Sept. 1989.

[7] S. I. Feldman, "Make—A program for maintaining software," *Software—Practice and Experience,* vol. 9, no. 3, pp. 255–265, 1979.

[8] C. W. Fraser and E. W. Myers, "An editor for revision control," *Ass. Comput. Mach. Trans. Programming Languages and Syst.,* vol. 9, no. 2, pp. 277–295, 1987.

[9] A. N. Habermann, "Automatic deletion of obsolete information," *J. Syst. and Software,* vol. 5, no. 2, pp. 145–154, May 1985.

[10] S. Horwitz, J. Prins, and T. Reps, "Integrating noninterfering versions of programs," *Ass. Comput. Mach. Trans. Programming Languages and Syst.,* vol. 11, no. 3, p. 345–387, July 1989.

[11] B. W. Lampson and E. E. Schmidt, "Organizing software in a distributed environment," in *SIGPLAN '83 Symp. Programming Language Issues in Software Systems,* San Francisco, CA, 1983; also in *SIGPLAN,* vol. 18, no. 6, pp. 1–13, June 1983.

[12] D. B. Leblang and R. P. Chase, Jr., "Computer-aided software engineering in a distributed workstation environment," in *Ass. Comput. Mach. SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environments,* Pittsburgh, PA, 1984; also in *SIGPLAN,* vol. 19, no. 5, pp. 104–112, May 1984.

[13] A. Mahler and A. Lampen, "An integrated toolset for engineering software configurations," in *Ass. Comput. Mach. SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environments,* Boston, MA, 1988; also in *SIGPLAN,* vol. 24, no. 2, pp. 191–200, Feb. 1989.

[14] ——, "Shape—A software configuration management tool," in *Int. Workshop on Software Version and Configuration Control,* Grassau, Germany, Jan. 1988.

[15] K. Marzullo and D. Wiebe, "Jasmine: A software system modelling facility," in *Ass. Comput. Mach. SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environments,* Palo Alto, CA, 1986; also in *SIGPLAN,* vol. 22, no. 1, pp. 121–130, Jan. 1987.

[16] E. W. Myers, "Efficient applicative data types," in *Proc. 11th Annual Ass. Comput. Mach. Symp. Principles of Programming Languages,* Salt Lake City, UT, 1984, pp. 66–75.

[17] D. Notkin, "The GANDALF project," *J. Syst. Software,* vol. 5, no. 2, pp. 91–106, May 1985.

[18] W. Obst, "Delta technique and string-to-string correction," in *1st European Software Engineering Conf.,* LNCS289, Strasbourg, France, Sept. 1987, pp. 64–68.

[19] D. Parnas, "Designing software for ease of extension and contraction," in *3rd Int. Conf. Software Engineering,* Atlanta, GA, May 1978, pp. 264–277.

[20] J. A. Plaice and W. W. Wadge, "A UNIX tool for managing reusable software components," *Software Practice and Experience,* 1993 (in press).

[21] ——, "Reducing the complexity of software configuration," in M. Tchuente, Ed., *Proc. 1st African Colloquium on Research in Computer Science,* Yaoundé, Cameroon, Oct. 1992, pp. 85–96.

[22] M. F. Rochkind, "The source code control system," *IEEE Trans. Software Eng.,* vol. SE-1, no. 4, pp. 364–370, Dec. 1975.

[23] E. E. Schmidt, "Controlling large software development in a distributed environment," Ph.D. dissertation, University of California, Berkeley, 1982.

[24] R. W. Schwanke and G. E. Kaiser, "Living with inconsistency in large systems," in *Int. Workshop on Software Version and Configuration Control,* Grassau, Germany, Jan. 1988, pp. 98–118.

[25] R. H. Thomason, Ed., *Formal Philosophy: Selected Papers by Richard Montague.* New Haven, CT: Yale University Press, 1974.

[26] W. F. Tichy, "Software development based on module interconnection," in *4th Int. Conf. Software Engineering,* Munich, Germany, Sept. 1979, pp. 29–41.

[27] ——, "Software development control based on system structure description," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1979.

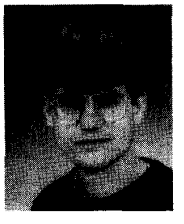
[28] ——, "Design, implementation, and evaluation of a revision control system," in *6th Int. Conf. Software Engineering,* Tokyo, Japan, Sept. 1982, pp. 58–67.

[29] ——, "The string-to-string correction problem with block moves," *Ass. Comput. Mach. Trans. Computer Syst.,* vol. 2, no. 4, pp. 309–321, Nov. 1984.

[30] ——, "RCS—A system for version control," *Software—Practice and Experience,* vol. 15, no. 7, pp. 637–654, 1985.

[31] ——, "Smart recompilation," *Ass. Comput. Mach. Trans. Programming Languages Syst.,* vol. 8, no. 3, pp. 273–291, July 1986.

[32] J. F. H. Winkler, "Report on the first international workshop on software version and configuration control," *SIGSOFT,* vol. 13, no. 4, pp. 61–73, Oct. 1988.



John Plaice received the B.Math. degree from the University of Waterloo, Waterloo, Ontario, Canada, and the DEA (master's) and Ph.D. degrees from the Institut National Polytechnique de Grenoble, Grenoble, France.

He is currently an Assistant Professor at the Université Laval, Quebec, Canada. His research interests cover all areas in which change and time play a role. He has recently published papers on real-time programming, dataflow programming, version control and software configuration.

Dr. Plaice is serving in 1993 as host for the Sixth International Symposium on Lucid and Intensional Programming.

William W. Wadge received the B.A. degree in mathematics from the University of British Columbia and the Ph.D. degree in mathematics from the University of California, Berkeley.

He is an Associate Professor of Computer Science at the University of Victoria, Victoria, B.C., Canada. His research interests include nonprocedural languages, dataflow, intensional logic, and data types.