Combination of Inheritance Hierarchies

Harold Ossher and William Harrison

IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598

Abstract

Making extensions to existing systems is a critically important activity in object-oriented programming. This paper proposes an approach in which extensions of all kinds are clearly separated from the *base hierarchy* upon which they are built, for ease of distribution and combination. Extensions, including extensions to existing classes, are written in separate, sparse *extension hierarchies*. The entire system is obtained by combining the extension hierarchies with the base hierarchy. Sequences of successive extensions can be combined using an *extension* operator, and parallel extensions can be combined using a *merge* operator, which might identify *conflicts* that must be reconciled. System building takes place at two levels: combining existing extensions from a library using these operators, and building new extensions when existing ones are not adequate. New extensions built in this way are added to the library, and so should be written to be as general and reusable as possible.

1 Introduction

The ability to make extensions with ease is one of the primary advantages of the object-oriented approach to building systems. Indeed, a common approach to building an object-oriented application is to extend a base system or library, or perhaps an existing application. We are in full agreement with Lieberman [7] that one wants a small extension to behavior to require just a small extension to code, and that adding new code is good, whereas modifying existing code is bad. We call this approach "extension-by-addition" [16].

Subclassing is a form of extension-by-addition, in which the extension is accomplished by adding entire classes. This paper addresses the issue of extending *existing* classes by addition, rather than adding new classes. Extending existing classes is important when:

- One wants objects created by existing code to exhibit some extended behavior. The creation code generally names specific classes; to obtain extended behavior without changing the creation code, one must extend those classes.
- One wants existing, persistent objects to exhibit some extended behavior. One must then extend the classes to which those objects belong.
- One wants instances of all existing subclasses of an existing class to exhibit some extended behavior.

The last case is similar to the problem identified by Lieberman of deriving a hierarchy of Colored-Shapes from an existing hierarchy of Shapes [7], except that his intent was to produce a separate hierarchy, not to modify the existing one.

The usual approach to extending existing classes is to edit the code associated with those classes, which, as noted above, is undesirable. The changes

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 ACM 0-89791-539-9/92/0010/0025...$1.50

made in this way become an integral part of the system, indistinguishable from the base system itself, or from other sets of changes. As many object-oriented programmers have discovered, this makes distribution of extensions or obtaining new releases of the base system extremely difficult. The bookkeeping aspects of this problem have been addressed by a variety of *change managers* [12, 14, 30, 33], effectively version management systems integrated with editors/browsers that keep track of changes made and are able to isolate them.

This paper proposes a linguistic approach to extension that keeps extensions of all kinds clearly separated from the base hierarchy. The idea is simple, but it has significant impact on the manner in which systems are built. Extensions are made in separate, usually sparse, inheritance hierarchies. The entire system is obtained by combining the extension hierarchies with the base hierarchy. Sequences of successive extensions can be combined using an *extension* operator, and parallel extensions can be combined using a *merge* operator. The base hierarchy can be replaced without changing the extension hierarchies. Merges and base hierarchy replacements can identify *conflicts* that must be reconciled.

Hierarchy combination can be applied to systems written in any object-oriented language. It is independent of the detailed semantics of inheritance, and so can be used in combination with any inheritance model.

Section 2 introduces hierarchy combination through a series of examples, and section 3 describes it formally. Section 4 briefly describes how systems can be built using hierarchy combination. Section 5 briefly discusses implementation approaches. Section 6 discusses change managers and other approaches to dealing with extensions, and how they relate to our approach. Section 7 discusses directions for future research.

2 Examples

Our approach to extension is to begin with a *base hierarchy*, and to extend it by developing a new and separate *extension hierarchy*. The extension hierarchy is usually sparse, containing just changed and new details of changed and new classes. It is then combined with the base hierarchy to form the complete system. Extension hierarchies can be created in sequence or in parallel, and can be combined in various ways.

Throughout this section we use simple examples drawn from the domain of vehicle simulations. The inheritance hierarchy sim_0 in fig. 1 supports very simple simulations, involving just moving vehicles and determining their positions. It will be used as the base hierarchy, upon which extensions are built.

Our first extension is to introduce the modeling of vehicle sales. This extension is specified in the extension hierarchy *sales* shown in fig. 1. We provide a single *sell* method associated with the *vehicle* class, to be inherited by its subclasses. Since the description of *car* is not changed by this extension, *car* does not appear at all in the extension hierarchy; the *car* class will, however, inherit the new vehicle *sell* method.

To accomplish the extension, the base and extension hierarchies are combined using the *hierarchy extension* operator, "$\triangleright$", as shown in fig. 2. In this simple situation, where no overriding occurs, the details of the classes are simply merged. In general, details in the extension hierarchy override corresponding details in the base hierarchy.

Fig. 3 shows another extension, *drawing*, to the vehicle hierarchy, this time to draw vehicles on a screen. Under the assumption that undifferentiated vehicles and cars will be drawn differently, this extension involves adding *draw* methods to both the *vehicle* and *car* classes. Note, however, that the superclass relationship between them is not changed, and therefore is not specified in the extension hierarchy (there is no vertical line between the classes in *drawing*).

The two extensions, *sales* and *drawing*, might

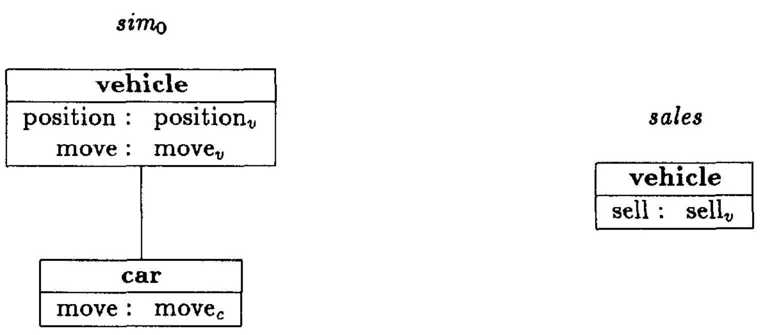


Figure 1: Base Hierarchy sim_0 and Extension Hierarchy $sales$

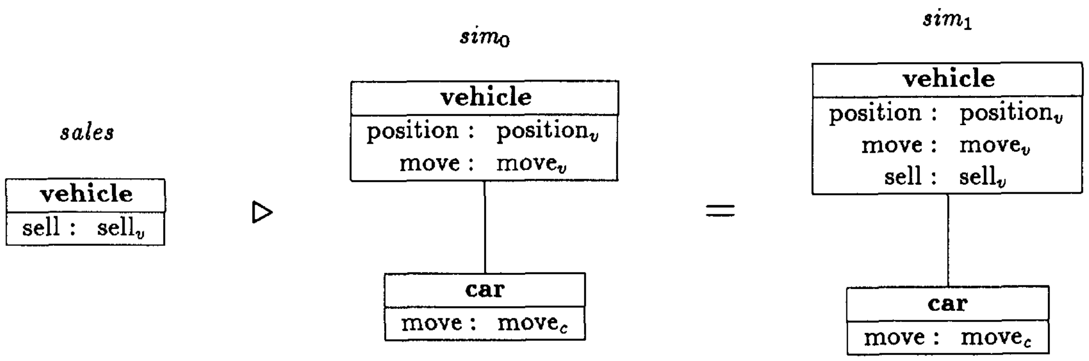


Figure 2: Hierarchy Extension: $sim_1 = sales \triangleright sim_0$

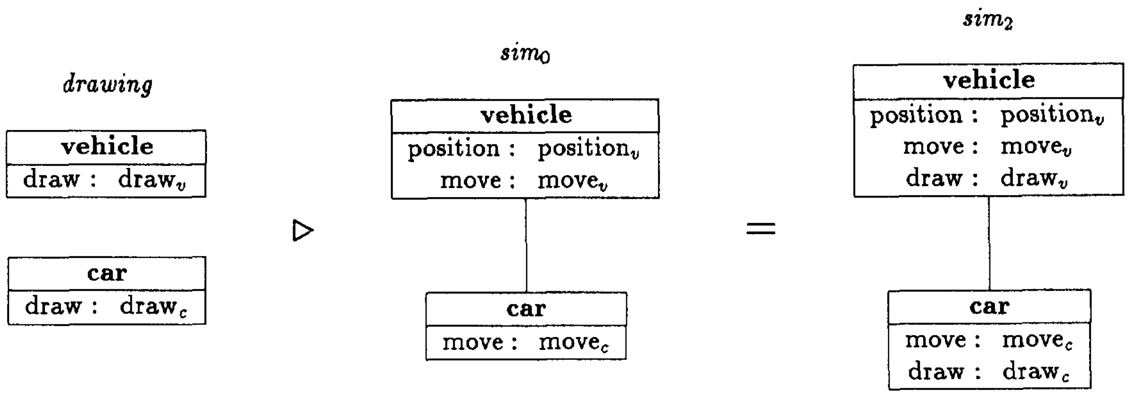


Figure 3: A Second Extension: $sim_2 = drawing \triangleright sim_0$

have been made in either order, or in parallel, by different programmers. It would make sense to integrate them, resulting in simulations capable both of handling sales of vehicles and of drawing vehicles on the screen. We wish to merge the extensions, and apply the merged extension to the base hierarchy sim_0. We express this as

$$(sales \diamond drawing) \triangleright sim_0$$

where the operator "$\diamond$" denotes *hierarchy merge.* The difference between extension and merge has to do with overriding. When one is extending a base hierarchy, it is reasonable to allow the extension to override some details of the base. Indeed, this is often how the extension is accomplished. When one is merging two independent extensions, however, one cannot let one override the other, for then some changes made deliberately will be lost. Merge is therefore defined to be the same as extension if there are no *conflicts*, but to fail if conflicts are present. This definition ensures that no separately-made changes are lost during a merge; it does not, however, guarantee that the separately-made changes will be *semantically compatible*, that is, that they will work correctly together. The issue of semantic compatibility of extensions is discussed in section 3.5.

In this example, there are no conflicts between *sales* and *drawing*: neither one contains any details that would override details in the other. The merge is shown in fig. 4, and its use as an extension of sim_0 in fig. 5.

Now suppose that after some use of the vehicle simulation hierarchy, including building of the *sales* and *drawing* extensions, a bug is discovered in $move_c$, the move method for cars. The bug can be repaired in an extension hierarchy, *bugfix*, shown at the left of fig. 6. It might seem unusual to repair a bug by adding an extension hierarchy rather than by simply editing the code. The advantage of doing so is that it is then possible to merge the change to the base hierarchy with the extensions made earlier to the base hierarchy. Inspection reveals that *bugfix* does not conflict with either of the earlier extension hierarchies, *sales* and *drawing*, so the merges succeed. It is therefore possible to form the following repaired hierarchies, the first of which was shown in the figure:

$$\begin{aligned} sim_0' &= bugfix \triangleright sim_0 \\ sim_1' &= sales \triangleright sim_0' \\ sim_2' &= drawing \triangleright sim_0' \\ sim_3' &= (sales \diamond drawing) \triangleright sim_0' \end{aligned}$$

3 Formal Description

This section presents a formal description of hierarchy combination. We begin with a simple formal model of inheritance hierarchies, just detailed enough to permit the definition of hierarchy combination.

3.1 Inheritance Hierarchies

An *inheritance hierarchy* consists of a set of named *class descriptions* organized into a superclass lattice. The information in these class descriptions, combined as dictated by the inheritance semantics, defines a set of *classes* bearing the same names.

Inheritance is usually discussed either by considering classes as types that specify objects as records that contain methods as well as other values [6], or by considering class descriptions that themselves contain the methods, as well as other details [9, 24, 31]. We take the latter approach, and model class descriptions simply as records of methods. This isolates the essentials, while ignoring issues such as class and instance variables. Variables can be handled by hierarchy combination in a manner analogous to the handling of methods, but they impose some requirements on the underlying language with regard to the manner in which object creation and initialization are specified.

A record can be thought of as a function from labels to field values [9]. Using this approach, we define a *class description*, d, as a function

$$d : \text{selectors} \rightarrow \text{methods}$$

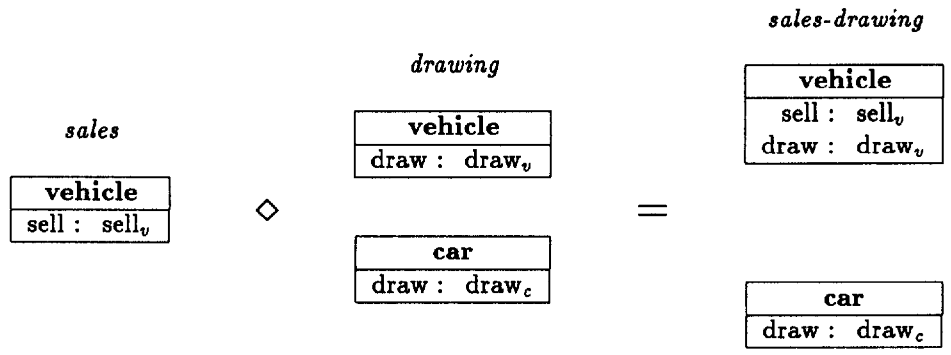


Figure 4: Hierarchy Merge: *sales-drawing* = *sales* ◇ *drawing*

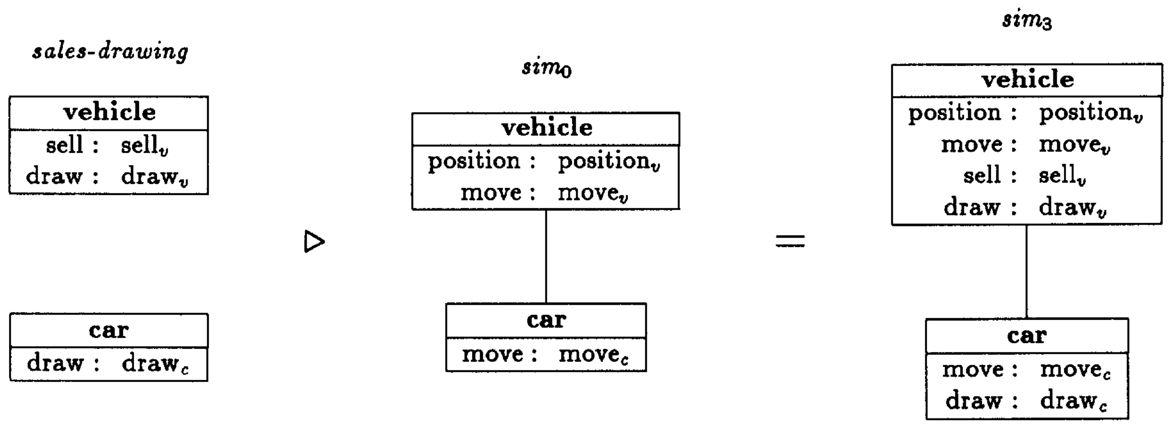


Figure 5: Merged Extensions: $sim_3 = sales\text{-}drawing \triangleright sim_0 = (sales \diamond drawing) \triangleright sim_0$

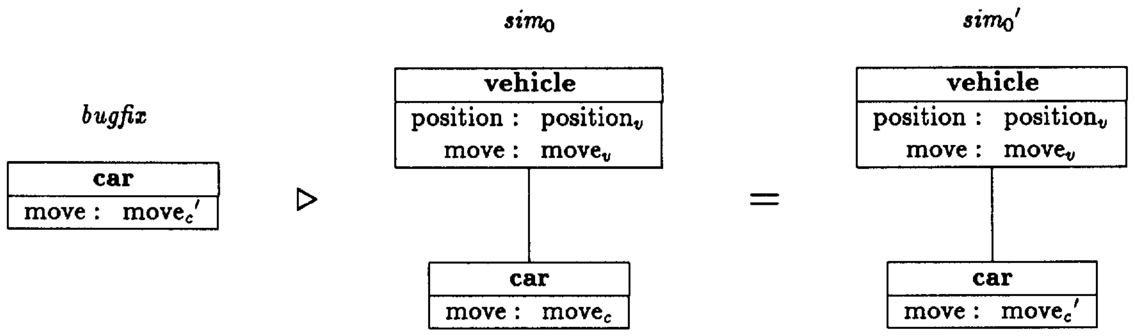
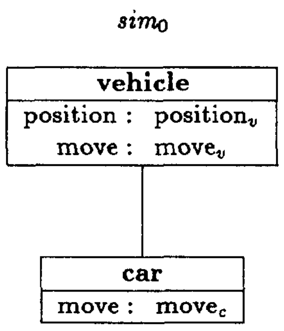


Figure 6: Fixing a Bug in a Base Hierarchy: $sim_0{}' = bugfix \triangleright sim_0$



$$\begin{aligned}
sim_0 &= (N_0, D_0, S_0)\\
N_0 &= \{\text{vehicle}, \text{car}\}\\
D_0 &= \{\text{vehicle} \mapsto \{\text{position} \mapsto \text{position}_v,\ \text{move} \mapsto \text{move}_v\},\\
&\qquad \text{car} \mapsto \{\text{move} \mapsto \text{move}_c\}\}\\
S_0 &= \{\text{vehicle} \mapsto \langle\rangle,\\
&\qquad \text{car} \mapsto \langle\text{vehicle}\rangle\}
\end{aligned}$$

Figure 7: A Simple Inheritance Hierarchy, sim_0

mapping selectors (operation names) to the corresponding methods. We define an *inheritance hierarchy* H as a triple (N, D, S), where N is a set of class names,

$$D : N \rightarrow \text{class-descriptions}$$

is the *class description function* specifying the description of each class, and

$$S : N \rightarrow seq(N)$$

is the *superclass function* specifying the sequence of immediate superclasses of each class. Sequences are used here rather than sets because superclass ordering is usually important in hierarchies involving multiple inheritance.

Both the class description function and the superclass function can be partial: in any particular hierarchy, either or both might be unspecified for a particular class. A class hierarchy in which both functions are total is termed *complete.* Only complete hierarchies are executable. Incomplete hierarchies can be combined to form complete hierarchies.

Example. The simple simulation base hierarchy shown previously in fig. 1 is shown again in fig. 7, with the components shown in detail. Though very simple, this hierarchy is complete.

Each class description in a complete inheritance hierarchy, together with the class descriptions of its superclasses and their ancestors, define a class. The details constitute the semantics of inheritance, and differ from language to language. For example, Smalltalk [13] has single inheritance, Beta [26] has prefixing, and CLOS [2], C++ [11] and Trellis/Owl [32] have multiple inheritance but with many differences in detail. Hierarchy combination does not depend on any specific inheritance semantics.

For the purposes of this paper, suffice it to say that inheritance in an object-oriented language is usually defined in terms of repeated application of a *class combination* operator, "$\oplus$", to the class descriptions in an ancestor sequence derived for each class from the superclass function. The standard definition of this operator is that methods supplied by the left operand override identically-named methods supplied by the right operand:

$$d_1 \oplus d_0 = d_1 \cup \{(s \mapsto m) \in d_0 | s \notin \text{domain}(d_1)\}$$

Some languages have more complex forms of class combination, such as supporting programmer-specified combination of methods [3, 28] or automatically combining clauses from inherited methods [10]. Some formal approaches to defining inheritance eliminate the linearization of ancestors, and involve a variety of operators [9, 8, 5, 4].

3.2 Hierarchy Extension

If H_0 and H_1 are inheritance hierarchies, then $H_1 \triangleright H_0$, read "H_1 extending H_0," is the inheritance hierarchy that results from applying the extensions in the *extension hierarchy* H_1 to the *base hierarchy* H_0.

Extension is defined formally as follows; an example was shown is fig. 2. Let $H_0 = (N_0, D_0, S_0)$

and $H_1 = (N_1, D_1, S_1)$. Then:

$$H_1 \triangleright H_0 = (N_1 \cup N_0,\ D_1 \triangleright D_0,\ S_1 \triangleright S_0)$$

The fact that the sets of class names are simply united implies that identically-named class descriptions in H_1 and H_0 are taken to refer to the same class. Such class descriptions are combined using class combination, mentioned above. $D_1 \triangleright D_0$ contains these combined class descriptions, and also all class descriptions associated with names that appear in just one of D_1 and D_0. Formally:

$$\begin{aligned} D_1 \triangleright D_0 = & \\ & \{ (n \mapsto d_1 \oplus d_0) \mid (n \mapsto d_1) \in D_1 \wedge \\ & \qquad\qquad\qquad (n \mapsto d_0) \in D_0 \ \} \ \cup \\ & \{ (n \mapsto d_1) \in D_1 \mid n \notin \text{domain}(D_0) \ \} \ \cup \\ & \{ (n \mapsto d_0) \in D_0 \mid n \notin \text{domain}(D_1) \ \} \end{aligned}$$

When hierarchy combination is used with a particular object-oriented language, the semantics of the class combination operator ("$\oplus$") are determined by that language. The definition of hierarchy extension imposes no requirements on them. As a result, the class descriptions in the combined hierarchy are just the same as could be obtained by subclassing within a single hierarchy. Instead of being separate subclasses, however, they are extensions of existing classes and retain the same names as the existing classes. New subclasses can, of course, be introduced in the extension hierarchy; they are included in the combined hierarchy by virtue of the second term in the union above:

$$\{ (n \mapsto d_1) \in D_1 \mid n \notin \text{domain}(D_0) \}$$

There are a number of possibilities for extension of superclass functions, involving merging of sequences, possibly taking the structure of the base and extension hierarchies into consideration. We take the simple approach that, for each class, the extension hierarchy totally overrides the base hierarchy:

$$S_1 \triangleright S_0 = S_1 \cup \{ (n \mapsto q) \in S_0 \mid n \notin \text{domain}(S_1) \}$$

Hence, if the extension hierarchy omits the superclass function for a class, it is taken to be the same as in the base hierarchy. This is the common case, since most enhancements to classes involve local changes rather than changes in the structure of the hierarchy. Such changes can be accomplished, however, by specifying the new superclass function for the class fully in the extension hierarchy.

It is important to note that an extension, though an addition of code, is not merely an addition of functionality: it can dramatically change existing functionality, especially if the superclass hierarchy is modified. If the base system H_0 produces a particular result when message m is sent to object o, the extended system $H_1 \triangleright H_0$ might produce a quite different result when m is sent to o. This is intentional. The primary advantage is that it facilitates the kind of extension so common in object-oriented systems, where code additions or changes are made deliberately to affect existing behavior. It does have the disadvantage that additions might inadvertently disrupt existing functionality when the intention is to add functionality. The hierarchy combination approach provides a suitable framework in which to investigate this issue, perhaps leading to the identification of classes of extensions that are guaranteed to preserve existing functionality.

3.3 Sequential Extensions

Repeated extension of an inheritance hierarchy leads to a sequence of hierarchies combined by means of the extension operator, such as

$$H_4 \triangleright H_3 \triangleright H_2 \triangleright H_1 \triangleright H_0$$

In some circumstances it might be valuable to maintain these hierarchies separately, in other circumstances it might be desirable to replace some or all of them by combined hierarchies. For example, if H_1 and H_2 are really considered to be extensions of the base system H_0 that will be part of a new release, and H_3 and H_4 realize two closely-related extensions that will be supplied as a package, it would make sense to compact the sequence to

$$(H_4 \triangleright H_3) \triangleright (H_2 \triangleright H_1 \triangleright H_0)$$

For such compaction to preserve semantics, one requires only that the extension operator "$\triangleright$" be associative. The definition given earlier implies that "$\triangleright$" is associative if and only if the class combination operator "$\oplus$" is associative. The standard class combination operator is, indeed, associative.

3.4 Parallel Extensions: Hierarchy Merge

If H_1 and H_2 are inheritance hierarchies, then $H_1 \diamond H_2$, read "H_1 merged with H_2," is the combined inheritance hierarchy incorporating the features of both H_1 and H_2. H_1 and H_2 might be parallel extensions of a common base, H_0, in which case the merged hierarchy will then be applied as an extension of H_0:

$$(H_1 \diamond H_2) \triangleright H_0$$

An example was shown in figs. 4 and 5.

To define the semantics of hierarchy merge, we first introduce the following notion: two inheritance hierarchies, H_1 and H_2, are *non-conflicting* if and only if

$$H_1 \triangleright H_2 = H_2 \triangleright H_1$$

Intuitively, this means that H_1 and H_2 can be combined without either overriding the other. In other words, all serializations of non-conflicting parallel extensions are equivalent. If H_1 and H_2 are non-conflicting, it is sensible to consider their merge as having the same meaning as their two equivalent sequential combinations. If H_1 and H_2 do conflict, however, parallel changes have been made to the same methods or to the superclass sequences of the same classes. We consider the merge of such hierarchies to be undefined, requiring that reconciliation of the conflicts occur before the merge is attempted. Hence:

$$H_1 \diamond H_2 = \begin{cases} H_1 \triangleright H_2 & \text{if } H_1 \triangleright H_2 = H_2 \triangleright H_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

3.5 Conflict, Interference and Semantic Compatibility

Non-conflict is different from "non-interference" as defined by Horwitz, Prins, Reps and Yang [22, 34]. Conflicting methods might not actually interfere, and hence might be amenable to their integration technique. Their definition of non-interference involves the base version as well as the extensions, so their technique would be applied to H_0, $H_1 \triangleright H_0$ and $H_2 \triangleright H_0$. As they point out, however, significant work remains to be done to extend their approach to realistic languages [22]. To work in the context of hierarchy combination, it would have to be able to cope with large, object-oriented systems, with their characteristic polymorphism and widespread (though encapsulated) updating and use of instance variables. In systems that manipulate persistent objects, the state of all such objects is of interest on termination. Conflicting methods, including methods that do interfere, can also be reconciled manually by a programmer familiar with both extensions and with what behavior is required.

On the other hand, two extensions that do not conflict might nonetheless interfere, in their uses of instance variables, their production of output, or their expectations about shared classes. For example, consider two extensions, one that adds a new subclass S_1 of class T_1, and another that adds a new subclass S_2 of class T_2. These are certainly non-conflicting extensions. Suppose, however, that code in S_1 assumes certain semantics for method m of T_2, but that S_2 overrides m with a method that has quite different semantics, as is allowed in many object-oriented languages. Since instances of S_2 are permissible wherever instances of T_2 are expected, S_1 might end up executing m on an instance of S_2, with disastrous results.

This particular problem can be circumvented by an approach, possibly enforced by formal specification and checking or by a construct such as *inner* in Beta [26], that requires an override method to be a true "semantic extension" of the method it is overriding. Even more subtle problems can occur, however. Suppose that code in both S_1 and S_2 set a particular instance variable to a value that is acceptable in terms of its type. The value set by S_1 might disrupt subsequent operation of S_2 methods, and vice versa. Even detailed invariants associated with instance variables will not necessarily trap all such problems.

These examples are abstract. Berlin described a real project in which various problems were experienced in integrating several object-oriented subsystems, and identified several dimensions of concern [1].

Interference among extensions can thus cause errors, but it can also be an important means for the extensions to cooperate. One therefore needs conditions different from non-conflict and non-interference to ensure that separate extensions will "work correctly together" when merged. We refer to extensions that have this ill-defined property as *semantically compatible.* Definition and exploration of semantic compatibility remain important areas for future research, discussed in section 7.

3.6 Ensuring Absence of Conflict

Knowing that extensions are non-conflicting is important, even if semantic compatibility is not guaranteed: the extensions can be merged in a simple, well-defined manner without any new code being overridden, as a first step towards full integration.

If hierarchy extension, "$\triangleright$", is commutative, then all extensions are automatically non-conflicting. This is a powerful argument for using commutative extension operators. The definitions given earlier imply, unfortunately, that "$\triangleright$" is not commutative: neither class combination, "$\oplus$", nor extension of superclass functions as defined is commutative. Alternative hierarchy extension operators are discussed in section 7.

If "$\triangleright$" is not commutative, the nature of specific extensions determines whether or not they conflict. We introduce a class of extensions, called *partitioned extensions*, that are guaranteed not to conflict.

The key intuition behind partitioned extensions is that extensions are accomplished by adding new code fragments rather than by changing existing code fragments (extension-by-addition), and that the new fragments are added in a new "space" so that fragments added by different extensions do not conflict.

Subclassing leads to partitioned extensions. If naming conventions or automatic renaming ensure that no two extensions add subclasses of the same name, the extensions cannot conflict: they are effectively in different name spaces. Similarly, adding methods with new selectors to existing classes leads to partitioned extensions.

Other extensions to existing classes usually cannot be accomplished as partitioned extensions, however. For example, enhancing an existing method to handle a new case involves changing the method itself. If two separate extensions make such changes to deal with two, separate, new cases, the changes conflict. In our work on extension-by-addition [16] we have developed three mechanisms that extend the kinds of changes that can be made as partitioned extensions:

- *Subdivisions* [18]. An operation can be declared to be subdivided on criteria other than the class of the object to which it is applied. Different methods can then be provided for the same class for each different value of the criteria. This is related to *multi-methods* in LOOPS and CLOS [3, 2], but extended to include discrimination based on non-parameter variables, and on *values* rather than types. Adding a new criterion, a new criterion value, or a method for a new criterion value is a partitioned extension; the new criterion or criterion value effectively defines a separate space for the methods. Enhancing a method to handle a new case can be accomplished as a partitioned extension by adding a new subdivision criterion value (of an existing or new subdivision criterion) to characterize the new case, and providing a new method for that specific case.

- *Structure-bound messages* [20]. A message type can be declared with a default routing. A structure-bound message sent to an object begins at that object and then propagates based on the default routing. "Interested" objects it encounters can explicitly handle it and/or reroute it. "Uninterested" objects need have no knowledge of it at all; it simply continues, following the default routing. Adding a new message type, or an override routing or handler for a new message type is a partitioned extension. The new message type effectively defines a separate space in which the additions take place.

- Our approach to instance variable accesses ensures that method code, even compiled method code, is independent of details of object layout [21]. This permits association of new instance variables with a class as a partitioned extension.

In our development of the RPDE^3 environment, many useful enhancements were be made as separately developed, partitioned extensions, and they were found to integrate easily and effectively [29].

Mechanisms such as those described above serve two important purposes. Firstly, they reduce the granularity of the code fragments needed to express extensions. For example, a specific subdivision of a method or a single instance variable declaration can appear in an extension hierarchy, without the need to copy, change and override the existing method or class definition. Secondly, they increase the number of extensions that can be accomplished as partitioned extensions. Mechanisms that do this are greatly to be encouraged because they increase the range of enhancements that can be accomplished with extensions that are guaranteed not to conflict. The syntax-directed program modularization feature of Beta [25] is interesting in this regard. It allows code fragments of arbitrarily fine granularity, determined by the programmer, to be treated as modules. This reduces the granularity of the fragments used to express extensions, but it does not lead to partitioned extensions because the fragments are not in new spaces; for an isolated fragment in an extension hierarchy to be useful, there must already be a reference to it in the base hierarchy. Extensions that involve overriding, or supplying definitions of symbols declared in the base hierarchy, cannot be guaranteed not to conflict.

It is worth noting that the integration achieved by combining partitioned extensions is loose. One is collecting together disjoint sets of code fragments that do not interfere with one another, but generally do not cooperate with one another either. The integrated system will include the functionality of each of the extensions separately. The extensions often complement one another other, however. For example, in RPDE^3 we integrated a structured program editor and support for hypertext links among objects, and obtained, without further effort, hypertext functionality within the editor. Once the integration is complete, one can write additional code fragments explicitly designed to exploit the presence of both extensions.

3.7 Replacement of a Base Hierarchy

Thus far we have considered building extensions upon an existing base hierarchy. Another problem that arises is replacement of the base hierarchy upon which an existing extension is built. This occurs primarily when a new release is obtained, either of an object-oriented system or library, or of some object-oriented application that one has extended. An example was shown in fig. 6.

Because of the properties of hierarchy extension discussed above, the following simple case covers all situations. Consider a base hierarchy H_0 and an extension hierarchy H_1, and assume that $H_1 \triangleright H_0$ is a correctly working application. We wish to replace H_0 with a new base hierarchy, H_0', and have the updated application $H_1 \triangleright H_0'$ continue to work.

Suppose that the new base hierarchy was built as an extension of the original one:

$$H_0' = H_e \triangleright H_0$$

This might at first seem unlikely, since most new releases involve bug fixes and deletion of code, as well as enhancements. However, bug fixes and deletion of code within methods can be realized as extensions that supply replacements for the methods concerned. Deletion of entire methods or classes cannot be realized as extensions as currently defined, but we believe that the definition could easily be extended to accommodate them. This remains a topic for future research.

Under the assumption that

$$H_0{}' = H_e \triangleright H_0$$

the base extension H_e and the application extension H_1 are parallel extensions, so we can apply the reasoning of the previous sections to them. If they are non-conflicting extensions, then the merged hierarchy

$$(H_1 \diamond H_e) \triangleright H_0 = H_1 \triangleright H_e \triangleright H_0$$

is well defined, and contains all the extensions made both in the new release of the base and in the original application. If they are semantically compatible extensions, then the application will work correctly upon the new base hierarchy.

4 System Building with Hierarchy Combination

The use of hierarchy combination leads to a new approach to system building, which involves two levels. At the higher level, existing base and extension hierarchies are combined using the extension and merge operators. The extensions are kept in a library, and the environment provides assistance with locating suitable extensions and checking that their combination is valid. This differs from traditional reuse situations in that one is composing sparse inheritance hierarchies that contain extensions to a variety of classes, rather than composing components that consist of distinct classes.

If no suitable extension exists in the library, one drops to the lower level: that of writing an individual extension. The goal here should be to write a new extension that can be added to the library, rather than a piece of special-purpose code that is of use only to the application at hand. Writing an extension is object-oriented programming, with the standard possibilities for reuse.

5 Implementation

Hierarchy combination can be implemented directly by language compilers or interpreters, by having them perform method resolution according to inheritance rules appropriate for a combination of hierarchies. It can also be used as the formal model specifying the semantics to be realized by change managers: the change managers perform the combination and present the compiler or interpreter with the single, combined hierarchy. We are currently building an implementation of hierarchy combination in the context of object-oriented tool integration [15].

Implementations of hierarchy combination based on change managers or based directly on most current compilers would require extensive compilation of the combined hierarchy. It is, however, possible to produce implementations in which the hierarchies involved in combinations (including method code) need not be available in their source form. This permits systems that are shipped object-code-only to be extensible nonetheless. The following are key requirements:

- Compiled dispatching information (the mapping from selectors and classes to methods) must be separated from compiled method code, and must be in a form that is suitable for combination. If this form is not suitable for use at runtime, there can also be an optimized form, but the unoptimized form must be retained, or be capable of reconstruction.

- Similarly, the compiled form of the hierarchy must contain information that controls the creation of objects in a form that can be combined. This is necessary to support combination of extensions that define new instance variables.

- Compiled method code must be independent of the offsets of instance variables. This is necessary so that multiple extensions that define new instance variables can be combined without the need to recompile the methods that operate on those instance variables. This issue, and our approach to dealing with it, are discussed elsewhere [21].

- Similarly, selectors must be encoded in the compiled method code in a way that is not dependent on a particular hierarchy, so that they remain valid in combined hierarchies. One approach is to use symbols derived from the selector names, possibly qualified by hierarchy name, and resolved by a linker after hierarchy combination.

Combination of compiled hierarchies then operates by combining the dispatching and instance-creation information according to the semantics of the combination operators, and linking this with the methods from all the combined hierarchies.

Persistent objects impose the additional requirement that existing instances be upgraded to include newly-defined instance variables, appropriately initialized. Our approach to this issue to to provide an *upgrade protocol* in the persistent store that detects outdated objects when they are brought into memory, creates space for the extra instance variables, initializes them to default values, and then calls "upgrade" methods on them, if such methods exist, to allow for programmer-specified initialization [21].

6 Alternative Approaches to Extension

The two most common approaches to extension in object-oriented languages are *subclassing* and code modification.

Using subclassing, an extension is accomplished by adding a new class, based upon some existing class(es) but having extended behavior. Subclassing is very much in the spirit of hierarchy combination: it is a restricted form of hierarchy extension in which all extensions consist of adding new classes. It does not support extension of existing classes, and so is not suitable when extended behavior must be exhibited by objects that already exist or by objects that are created by code that already exists.

It is possible to extend an existing class by adding a new class as its *superclass*. This involves more than merely adding the new class, however. It changes the structure of the inheritance hierarchy at an internal node, a type of change that is usually poorly supported by object-oriented systems. It also requires editing of the original class to reflect the new superclass, and probably also to remove some of the methods or to move them to the new superclass.

For languages, such as Smalltalk [13], that require all methods for a class to be packaged into a single class description, all changes to an existing class involve modifying the class description. *Change managers* have been developed to alleviate the primary problem with this approach: the fact that changes become mingled with the base system and with one another. An early and excellent example is PIE [14]. It represented Smalltalk programs as networks of objects that could be edited. Changes were always made in a new *layer*. Layers could be saved, and later installed either by the same user or any other with a compatible base system. A sequence of layers, called a *context*, could be installed, with later layers dominating earlier ones. PIE provided interfaces for examining multiple contexts and integrating their changes. The layers and contexts thus served to identify and isolate changes, and to facilitate their distribution. A number of change managers for Smalltalk have been developed since PIE, such as the Smalltalk-80 *project* mechanism with associated change-set and change-management browsers [12], the version handler described by Putz [30], and Orwell [33].

Some object-oriented languages, such as Flavors [28] and CLOS [2], do not require all methods for a class to be packaged into a single class description. An extension that can be written as a collection of method definitions for a variety of classes can be packaged into a file for distribution. This approach thus supports separation of extensions consisting of new or changed methods. It cannot support separation of extensions involving addition of instance variables or changes to the superclass structure, however.

System building in C++ [11] is usually done by building an application from scratch, probably using and specializing classes in a library. Subclassing is adequate in this context, because there are no existing references to classes and no existing instances. However, if such applications are later to be enhanced, or to be used as bases for new applications, subclassing becomes inadequate.

The Beta language supports patterns that can consist of one or more classes, and that can be nested [26, 27]. A hierarchy can be thought of as such a pattern, with individual classes nested within it. One module can inherit from another, extending those classes it wishes to change. Cook proposed a similar approach, which he called "hierarchy inheritance" [8]. Bracha touched on this issue also, but did not support it fully in Jigsaw because of a desire to do all type checking statically [4]. Support for hierarchy combination in these languages has not been explored in detail, and where it is mentioned the emphasis is on deriving one hierarchy from another, as in Liberman's ColoredShape example, rather than on extending the behavior of existing classes and instances.

7 Future Research

Our approach to extension raises a number of interesting and important areas of future research, including:

- Techniques for reducing conflicts between parallel extensions, including commutative extension operators.
- Formal definition of the notion of semantic compatibility.
- Identification of classes of extensions or extension mechanisms that will result in semantic compatibility, or at least that are likely to do so.
- Formalizing the requirements that an extension places on its base.

Each of these is discussed briefly in this section.

Parallel extensions that conflict cannot be integrated simply. It is therefore important to find techniques for resolving or reducing conflicts. Yang, Horwitz and Reps use of dependency analysis as an aid to automatic integration [34] is a conflict resolution approach. Conflicting procedures can be integrated automatically provided the algorithm can show that they do not interfere in their use of variables. Mechanisms such as subdivision and structure-bound messages described in section 3.4 are conflict reduction techniques. They enlarge the set of enhancements that can be accomplished by means of partitioned extensions, which are necessarily non-conflicting. Discovery of other mechanisms with this property, or of other classes of extensions that are necessarily non-conflicting, would enlarge this set further.

An especially powerful conflict reduction approach is to use commutative extension operators. Indeed, as noted earlier, this eliminates conflicts entirely. Two changes must be made to achieve this: both class combination, "$\oplus$", and superclass function extension must become commutative. Essentially, the way to do this is to eliminate overriding and replace it with commutative combination. For example, a method could consist of a set of fragments, all of which are to be executed. Combination of such methods would then be the union of the sets. To ensure commutativity, the execution order of the fragments must not be dictated by the order of the operands. This approach was followed in Meld [23], where methods consist of equations and rules that are evaluated in an order dictated by inherent dependencies, and in OOLP [10], where all methods provided by superclasses are evaluated in arbitrary order. It is not clear that this approach can work effectively if methods are sequential code. In general, the challenge is to find commutative combination operators that permit effective and natural writing of methods.

Though lack of conflicts is important, the property that really matters in integrating parallel extensions is semantic compatibility: the property that the extensions will "work correctly together." Clearly this needs formal definition. Yang, Horwitz and Reps work [34] addresses this in-the-small, essentially defining semantic compatibility to mean non-interference in use of variables. On a larger scale, however, separate extensions might well modify the same variables, causing the integration to behave differently from either of the extensions separately. Such behavior is often exactly what is desired. It is necessary to find a way to express the "correct" behavior even in such cases. We expect it to involve specification of required semantics, not just analysis of existing semantics.

Even when a definition of semantic compatibility is found, it will probably be more useful from a practical point of view to introduce the notion of *syntactic compatibility*: a property that can be checked easily and that implies, or at least is likely to imply, semantic compatibility. For example, partitioned extensions are weakly syntactically compatible: if they are written carefully using modern notions of encapsulation, there is reasonable likelihood that they will be semantically compatible. This assertion is borne out by experience [29], but there is no guarantee. Determining how to increase the likelihood of semantic compatibility, even if it cannot be assured, is important.

Characterizing the requirements that an extension places on its base is also important. Even when one writes an extension with a particular base in mind, one wishes to have confidence that the extension enhances the base and does not disrupt it. This issue is similar to semantic compatibility among parallel extensions, discussed above. When one selects an extension from a library and wishes to apply it to a base other than that for which it was written, one also has to ensure that the base provides what the extension relies upon. Dealing with this issue involves formalizing what an extension requires of its base, and being able to check these requirements against specific bases. A related problem is that of finding in the library extensions that are appropriate to one's base and one's needs.

8 Summary and Conclusions

We have introduced hierarchy combination as a new approach to object-oriented system building. Instead of building systems by editing existing systems or by selecting and specializing classes from a library, one builds systems by writing separate, sparse extension hierarchies that enhance an existing base. The extension hierarchies and the base hierarchy are combined to form the complete system, using extension and merge operators. One can build up a library of extensions, and then build systems by selecting extensions from the library and combining them.

The advantages of the approach include:

- It supports extension of existing classes and changes of the superclass structure, as well as addition of new classes. Extension of existing classes includes introducing new instance variables and methods, and overriding existing methods.

- It keeps extensions well separated, facilitating distribution of extensions and selection of desired extensions from a library.

- In the absence of conflicts, it supports integration of separately developed extensions and upgrading of the base hierarchy.

- It is a linguistic approach with precise but easily-understood semantics. This aids programmer understanding and also provides a framework within which to investigate semantic issues such as semantic compatibility and extensions that preserve existing functionality.

- It depends only on a few standard properties of inheritance hierarchies, such as class names, descriptions and superclasses, and is orthogonal to such details as inheritance semantics or class or method combination semantics. It is therefore applicable to object-oriented languages in general.

- Implementations can support combination of hierarchies whose source form is not available.

Informal use of hierarchy combination in the development of RPDE^3 has shown that significant extensions can be written separately, and in a way that makes them easy to integrate [29]. In our continuing work on the CLORIS object-oriented database [19], the PlusPlus object definition environment [17] and object-oriented tool integration services [15], we plan to provide language and environment support for hierarchy combination as a vehicle for further exploration and validation.

References

[1] Lucy Berlin. When objects collide: Experiences with reusing multiple class hierarchies. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 181–193, Ottawa, October 1990. ACM.

[2] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification X3J13. *SIGPLAN Notices*, 23, September 1988.

[3] Daniel Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 17–29, Portland, September 1986. ACM.

[4] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.

[5] Gilad Bracha and William Cook. Mixin-based inheritance. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 303–311, Ottawa, October 1990. ACM.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstractions and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[7] Steve Cook. Panel P2: Varieties of inheritance. In *OOPSLA 87 Addendum to the Proceedings*, pages 35–40, Florida, October 1987. ACM.

[8] William Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown Univerisity, 1989.

[9] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 433–443, New Orleans, October 1989. ACM.

[10] Mukesh Dalal and Dipayan Gangopadhyay. OOLP: A translation approach to object-oriented logic programming. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD89)*, pages 593–606. North-Holland Physics Publishing, December 1989.

[11] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[12] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984. Chapters 4 and 23.

[13] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[14] I. P. Goldstein and D. G. Bobrow. A layered approach to software design. Technical Report CSL-80-5, Xerox Palo Alto Research Center, December 1980.

[15] William Harrison, Mansour Kavianpour, and Harold Ossher. Integrating coarse-grained and fine-grained tool integration. In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering (CASE '92)*, Montreal, July 1992. To appear.

[16] William Harrison and Harold Ossher. Extension-by-addition: Building extensible software. Research Report RC 16127, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 1990.

[17] William Harrison and Harold Ossher. The PlusPlus object definition environment. Research Report RC 16283, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 1990.

[18] William Harrison and Harold Ossher. Subdivided procedures: A language extension supporting extensible programming. In *Proceedings of the 1990*

International Conference on Computer Languages, pages 190–197, New Orleans, March 1990. IEEE.

[19] William Harrison and Harold Ossher. CLORIS: A clustered object-relational information store. Research Report RC 16723, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 1991.

[20] William Harrison and Harold Ossher. Structure-bound messages: Separating navigation from processing. Research Report RC 15539 Revised, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, October 1991.

[21] William Harrison and Harold Ossher. Attaching instance variables to method realizations instead of classes. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 291–299, Oakland, April 1992. IEEE.

[22] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[23] Gail E. Kaiser and David Garlan. MELDing dataflow and object-oriented programming. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 254–267, Orlando, October 1987. ACM.

[24] Samuel Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Conference Record of the Fifteenth Annual Symposium on Principles of Programming Languages*, pages 80–87, San Diego, January 1984. ACM.

[25] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Syntax directed program modularization. In P. Degano and E. Sandewall, editors, *Integrated Interactive Computing Systems*, pages 207–219. North-Holland, 1983.

[26] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA porgramming langauage. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[27] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 397–406, New Orleans, October 1989. ACM.

[28] David A. Moon. Object-oriented programming with *Flavors*. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–8, Portland, September 1986. ACM.

[29] Harold Ossher and William Harrison. Support for change in RPDE^3. In *Proceedings of the Fourth Symposium on Software Development Environments (SDE4)*, pages 218–228, Irvine, December 1990. ACM SIGSOFT.

[30] Steve Putz. Managing the evolution of Smalltalk-80 systems. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 273–286. Addison-Wesley, 1983.

[31] Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Conference on Lisp and Functional Programming*, pages 289–297. ACM, 1988.

[32] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Killian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, Portland, September 1986. ACM.

[33] Dave Thomas and Kent Johnson. Orwell: A configuration management system for team programming. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 135–141, San Diego, September 1988. ACM.

[34] Wuu Yang, Susan Horwitz, and Thomas Reps. A program integration algorithm that accommodates semantics-preserving transformations. In *Proceedings of the Fourth Symposium on Software Development Environments (SDE4)*, pages 133–143, Irvine, December 1990. ACM SIGSOFT.