Environmental Acquisition — A New Inheritance-Like Abstraction Mechanism

JOSEPH GIL DAVID H. LORENZ
The Faculty of Computer Science,
Technion—Israel Institute of Technology,
Technion City, Haifa 32000, ISRAEL;
Email: { yogi | david } @CS.Technion.AC.IL

Abstract

The class of an object is not necessarily the only determiner of its runtime behaviour. Often it is necessary to have an object behave differently depending upon the other objects to which it is connected. However, as it currently stands, object-oriented programming provides no support for this concept, and little recognition of its role in common, practical programming situations. This paper investigates a new programming paradigm, *environmental acquisition* in the context of *object aggregation*, in which objects acquire behaviour from their current containers at runtime. The key idea is that the behaviour of a component may depend upon its enclosing composite(s). In particular, we propose a form of feature sharing in which an object "*inherits*" features from the classes of objects in its environment. By examining the declaration of classes, it is possible to determine which kinds of classes *may* contain a component, and which components *must* be contained in a given kind of composite. These relationships are the basis for language constructs that supports acquisition. We develop the theory of acquisition that includes topics such as the kinds of links along which acquisition may occur, and the behaviour of routine (methods) and attribute features under acquisition. The proposed model for acquisition as a hierarchical abstraction mechanism is a strongly typed model that allows static type checking of programs exploiting this mechanism. We compare it to several other mechanisms including inheritance and delegation, and show that it is significantly different than these.

OOPSLA '96 CA, USA
© 1996 ACM 0-89791-788-X/96/0010...$3.50

1 Introduction

"Nature vs. Nurture?"[1] This long standing question obsessed philosophers, psychologists, laymen and even physicists [46] for years. The dispute is over the relative importance of heredity and environment in determining the makeup of an organism. However, for the object-orienteer[2], it has rarely been a problem: The basic character of an object, sometimes called "behaviour" in the object-oriented (OO) jargon, is determined at birth (instantiation), and not by the household (the composite object) of which it is a part. This simplistic sweeping claim is, as are all such claims, false for humans. How true is it for objects? How should the "nurturing" of an object affect its manners? Can such influence be dealt with in a (type) safe manner?

In this paper we address these questions. We explain what is and what is not "the influencing environment" of an object. We show that there are many important cases, both in the problem and program domains, in which the need for *environmental affect* naturally arises. We then propose a new abstraction mechanism, *environmental acquisition*, and study its possible realization in a strongly-typed programming language.

This paper does not provide solutions to all the problems it raises. Rather, it presents a framework for addressing the issues involved in the many aspects of environmental affects.

1.1 Motivation

Consider the following example which may occur in an automobile industry application: An object of a class Car depicted in Figure 1, is a *composite* which comprises *components* such as objects of class Door. Suppose that it is known that a car is coloured red, then we are likely

[1] Also known as the environmental-heredity controversy.
[2] or, should we write, object-orientalist?

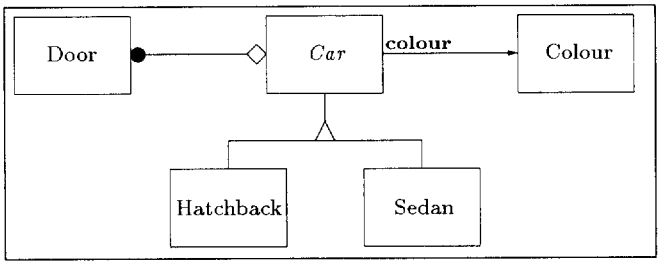

Figure 1: Acquisition in the problem space

to infer that its doors are red as well. However, although the door "inherits" its colour from the car of which it is part, it would be wrong to derive Door from Car. This "colour-inheritance" is related to the "in-a" link which binds doors to cars.

Consider now the subclasses of Car, say Sedan and Hatchback, which are distinguished among other things by their number of doors. Class Door "inherits" its colour from Sedans and Hatchbacks alike, just as it would from another hypothetical class Airplane which does not stand in an "is-a" relationship with Car. We call this kind of inheritance *environmental acquisition* (acquisition for short) and distinguish it from inheritance. Observe that since acquisition binds objects and not their classes, it does not (indeed it cannot) induce any subtype relationship. This is in contrast to the common role of inheritance in programming.

The above example is drawn from the problem space. Another example, which belongs in the program space, is that of class objects. In a pure object-oriented programming (OOP) model, such as that of SMALLTALK [19], classes are also objects. The concept of class objects occurs even in less pure models such as Objective-C [12], and SOM [15]. A class object is an instance of a meta-class Class.[3] It provides a mould for the instantiation of objects in the class and defines their behaviour. If a class C_1 inherits from C_2, then the class object of C_1 has a "super" link to its *containing* class object of C_2, as shown in Figure 2. Inheritance is realized by propagation of features[4] of C_2 along that pointer from C_1. This propagation can be thought of as acquisition in the meta-level. Part of the role of the meta-class Class is to define and implement the process of this propagation.

The fact that the propagation of features across links is not part of the usual object model contributes to the complexity of understanding and programming with class objects and meta-classes. Assuming a single-inheritance scheme, a view which mitigates this difficulty is that of class objects as representing sets. The class object Car models the set of all cars. The class objects Sedan and Hatchback model their corresponding sets, which are *part-of* the class object of Car. By allowing components to acquire attributes from their respective composites we put "colour-inheritance" of doors from cars, and the acquisition of features of the class object Sedan from the class object of Car at at the same conceptual level.

1.2 Acquisition vs. inheritance

Generally, an OO system encompasses two hierarchies, as depicted in Figure 3: an "is-a"-inheritance hierarchy of *classes*, and an an "in-a"-composition hierarchy of *objects* where acquisition dwells. The two hierarchies are tied by "instance-of" links between objects and classes which are drawn as dotted lines in the figure. Despite superficial similarity, there are important differences between inheritance and acquisition. These are highlighted by making the distinction between shared and particular features of an object. *Shared features* are those which are determined by the object's class. They include *behaviour* (declaration and definition of methods) and *structure* (declaration of instance variables). *Particular features* may be different in different objects of the same class. They include the object's identity and *state* (current values of instance variables). Inheritance pertains to classes and therefore serves as a means of abstraction over shared features. In contrast, acquisition can be viewed as a means of abstraction over particular features.[5]

Let $a_1, \ldots, a_n$ be objects of classes $C_1, \ldots, C_n$ respectively, such that a_1 is an instance variable of a_2, and so on, as depicted in Figure 4. All shared attributes of a_1 are determined by C_1. Traditionally, the particular features of a_1 are independent of both the shared and the particular features of a_2. *Environmental affect* on a_1 is tantamount to the dependence of its particular features on the class C_2 and its instance a_2. More generally, the *environment* of a_1 consists of the particular features of $a_2, \ldots, a_n$ as well as their shared features $(C_2, \ldots, C_n)$. The environmental affect on a_1 is the extent to which its particular attributes depend on the environment.

Unlike inheritance, acquisition does not impose subtyping: a Door is not a Car just as Clyde the elephant [13] is not the jungle it resides in. In subtyp-

[3] For the purposes of the example, it is sufficient to assume that there is only one meta-class and that abstraction stops at Class. That is, there are no meta-meta-class, meta-meta-meta-class etc. However, the example becomes even more interesting, albeit more complex, if these are allowed.

[4] Here, and henceforth, our terminology adheres as much as possible to that of EIFFEL [32].

[5] One may argue that in the case of inheritance an object inherits particular features from its subobjects. With acquisition the "subobject" would had inherited from the containing object.

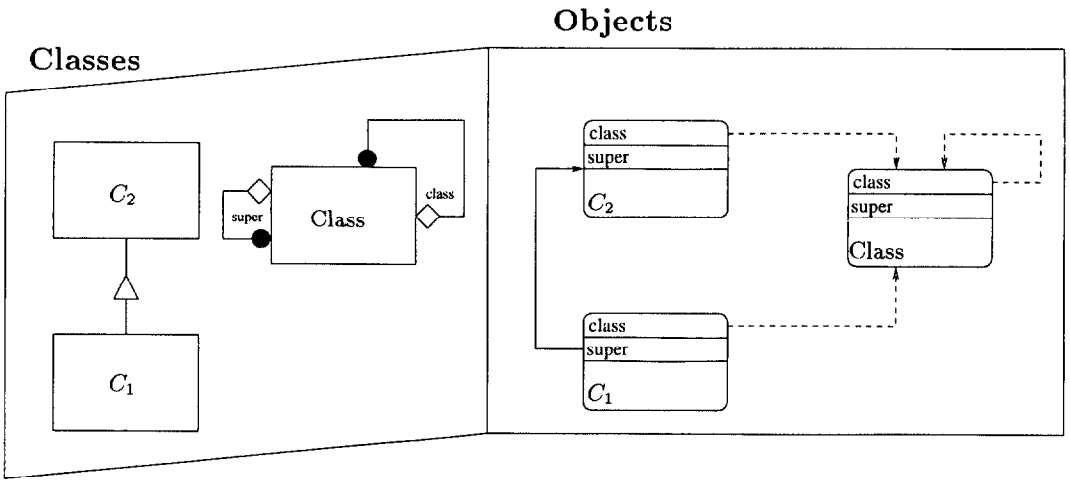

Figure 2: Meta class acquisition in the program space

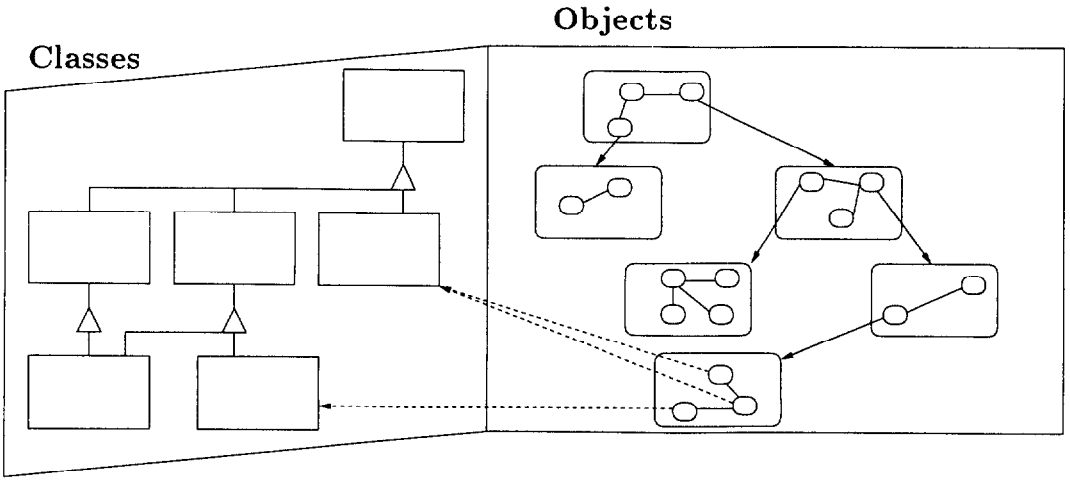

Figure 3: The "is-a"-inheritance and the "in-a"-composition hierarchies

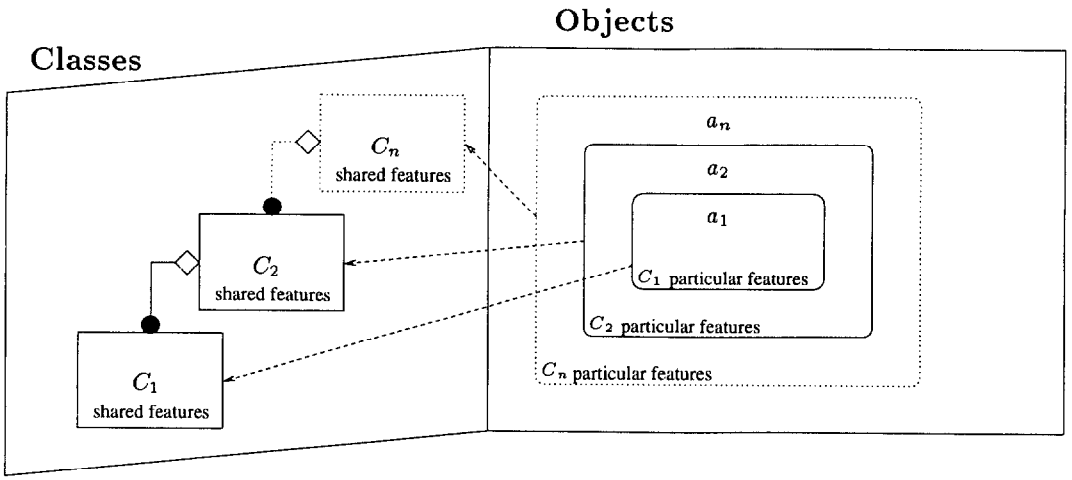

Figure 4: Environmental affect on a_1

ing, the heir could be used wherever the parent is used. Hence, all features of the parent *must* also exist in the heir. With acquisition, this is unexpected, often undesirable, and in some cases impossible: Although the behaviour of Clyde in the jungle might be different than in a zoo, Clyde is not green just because the jungle is. On a workstation, we may want our windows to acquire the colour palette, but not the exact colour of their composite window. In word processors, words cannot be justified, while the paragraphs they are in can.

Another difference between acquisition and inheritance is therefore that acquisition is done on a feature by feature basis. The acquiring class must enumerate all features it needs to acquire. In offering this freedom of selection, acquisition is more expressive than inheritance. The penalty, that comes in the form of long lists of such enumerations, can be alleviated with semantic grouping of features, or even collecting features that are acquired together in a compound feature class.

Our proposal for acquisition is strongly typed; specific ways are provided for the programmer of an object to cater for all of its potential environments and deal in a type safe manner with a concrete environment in which it exists. Moreover, a component can be dynamically moved to a different composite, resembling perhaps dynamic inheritance. A consequence of this is an inclination towards forwarding rather than delegation semantics.

Acquisition is also subject to polymorphism. *Environmental polymorphism* means that there are many possible variant behaviours of objects of a given class, and that the precise behaviour is dependent on the environmental affect. The terms environmental acquisition and environmental polymorphism are complementary, just as inheritance and *subtype polymorphism* [41, 8] are. Environmental polymorphism is different from subtype polymorphism: With subtype polymorphism, the code applicable to a certain type (class), be it part of that type definition or not, is also applicable to all of its subtypes (subclasses). The code's behaviour is therefore parameterized by the actual object it operates on. With environmental polymorphism, the object's behaviour is parameterized by its surrounding environment.

Acquisition is different from programming with exemplars [29, 5] in the same way that classical class-based languages are different from languages such as SELF[6] [47]. Although acquisition can be imitated (to a known extent) by the delegation mechanism, such an imitation does not match a disciplined use of this paradigm as enforced by built-in lingual support. This difference is similar to that of high-level languages to assembly language. Although machine code is at least as powerful as high-level languages, since it implements them all, the availability of high-level promotes better programming practices. For example, as we shall see, it is possible to practice type-safe programming with acquisition, a trait which the unharnessed power of delegation excludes.

[6] There is no need in a "single-hierarchy" system with only one kind of objects, as in SELF, for meta-classes because objects describe themselves. SELF provides, however, dynamic inheritance in the form of delegation.

Outline The rest of the paper is organized as follows: To give the reader a taste of the motivation for this research, we present in Section 2 several important application domains in which acquisition emerges naturally. Sections 3, 4 and 5 together develop the theory of acquisition: Section 3 deals with the channels along which acquisition occurs, Section 4 with environmental polymorphism, and Section 5 deals with the question of propagating features in these channels in a type safe manner. In Section 6 the proposed acquisition is compared with few other models, including dynamic inheritance, delegation, genericity, composite object support [23] and complex associations [27]. Finally, Section 7 gives the conclusions and possible directions for further research.

2 Application Domains

Containment hierarchy is a ubiquitous concept in programming methodology in general, and specifically in the OO paradigm. The COMPOSITE design pattern [16] appears in almost any OO system. Many of the OO analysis and design methods even devote a special notation for containment [3, 10, 38]. In this section we show that there are abundant cases of containment hierarchies where contained objects have different behaviours depending on their surrounding environment. We give five application domains in which this phenomenon occurs naturally.

2.1 GUI Systems

OO graphic user interface (GUI) frameworks typically organize screen elements: windows, views, dialog boxes, and so on, in a view tree. Systems of this sort are for example, Turbo-Vision and its descendent OWL [4, 43], InterViews [30], and that of NeXT [45]. Responsibilities, such as screen drawing and handling input events, are distributed down this hierarchical view-tree. A screen element to which responsibility is delegated from its (environmental) parents, also acquires some of its

parents' traits. In Turbo Vision [4] for example, we see the following acquired traits.

Origin of the coordinate system. The coordinate system of a screen element is relative to that of its parent.

Error handling. The routine to call in case an unrecognized event occurs may be that of the parent.

Control flow. Modal screen elements are subtrees which, when activated, disable all elements external to them. Examples of modals are yes-no message boxes and the application itself. When a modal element terminates, it returns control to its nearest enclosing modal element. The nearest enclosing modal element is implemented as an environmentally-acquired attribute which exists in all screen elements.

Attributes palette. Screen colours (aka attributes) of a screen element are given as indices of a palette table stored in one of the screen element ancestors. The values stored in this table serve in turn as indices to an ancestor of this ancestor and so on until the application global attributes are reached. This acquisition mechanism is designed for flexibility and power, but it makes programming and reasoning with palettes a complex task.

InterViews [30] uses acquisition for computing constraints on an object's size. Other GUI systems use acquisition for other purposes e.g., context sensitive help, where a screen element acquires its response to a help request from its enclosing element but may override it to support more specific help.

These GUI systems are written in languages that do not support acquisition as an environmental inheritance mechanism. Acquisition must be therefore emulated, usually by a complicated web of pointers and schemes for call-back. Beyond unwieldy complexity, this results in difficult to understand features (as in the attribute palettes example), or in non-safe programming, (e.g., is it always guaranteed that an existing modal will find an enclosing modal?).

2.2 Graphic modeling

The intricacies of Turbo Vision's attribute palette are only multiplied in the realm of high resolution graphics modeling 3-D objects. Here it is necessary to address issues of acquisition of many more kinds of attributes such as line styles, colour, shading, textures, modeling type, and transformation matrix, spread along very complex objects. The graphics community recognizes that the ensuing questions are difficult [14, Section 9.2]. However, there is at least one major graphic standard which includes acquisition [37].

2.3 Text processing

Emphasized text in LaTeX [28] normally prints in *italics.* However, *an emphasized* within *an emphasized block prints in* roman *(as demonstrated in this sentence).* This is only one of the many examples in the domain of desktop publishing systems where the behaviour of text elements is strongly dependent on their surrounding environment. Systematic approaches to document processing, such as SGML [20, 7] and RTF (Rich Text Format), use a hierarchical representation, and let attributes such as type-face, text-size, and bounding boxes be environmentally-inherited by elements from their surrounding elements. Some modern word-processors (e.g., Dagesh [1]) even explicitly use the word "inheritance" to denote what we call acquisition.

The usage of acquisition is very evident in the scoping model of TeX [25]: the value of all macros, declarations, and registers, is acquired at each point from the innermost enclosing scope in which they are defined. Acquisition and environmental polymorphism are part of the reason why programming TeX macros is so notoriously difficult. A macro is a polymorphic object whose behaviour depends on the values of the commands and on other macros that it calls at the time of activation. To add to the complication, a macro may change its own definition during its execution. Therefore, recursive calls may amount to something totally different than our usual understanding of recursion. Better understanding, strong typing, and disciplined acquisition should make TeX macros less awkward.

2.4 User defaults in an operating environment

Fancy GUI windowing environments allow the attachment of various defaults and user preferences to files and other resources. For example, one may attach a word processing application, customized and configured appropriately, to a certain type of document. On clicking on a document's icon, the corresponding editing application will be invoked. If the user manipulates many different projects, or if a large multi-user environment is to be supported, this attachment has to be carried out in an orderly manner.

Even in a simple **Unix** environment, a default system is needed: Different users have different preferences, which may further depend on the types of files and their location. Upon editing a file of an unknown type located anywhere under the `Programs` directory, the editor should be in programming mode. If the same file is under a `Documents` directory, defaults should change

accordingly. This behaviour may be different depending under which home directory these directories reside.

Acquisition, together with an inheritance hierarchy of types of files and other resources, is the appropriate way of setting defaults. Its absence brings about a major source of confusion for naive users and a source of headache for the system administrator. Current solutions in **Unix** use a tangled mixture of compile-time flags for applications, system-global, user and directory initialization files (.Xrc in the **Unix** jargon), together with environment variables settings. Shared (networked) installation of many applications in MS-Windows is next to impossible.

2.5 Language processors and reverse engineering

Processing formal (programming) languages is another application domain which calls for acquisition: compilers, interpreters, automatic generation of test cases, computing metrics, etc. If the processed language belongs in the Algol family and has static binding, then acquisition is applicable. A parse tree for a specific input program of this language only captures the syntactical aspects of the program [17]. The semantic information can be computed from that tree with the help of acquisition. Here are a few examples, all taken from C++.

Constructors. Constructors and ordinary member functions have essentially the same syntax, but generate quite a different code. The precise type of method can only be determined by the name property of the enclosing `class/struct` definition.

Variable definition vs. function declaration. The `x y(z);` C++ statement is either a function declaration or a variable declaration, depending on the declarations of `x`, `y` and `z`. These declarations can only be found in the environment surrounding the statement.

Members' visibility. The visibility (`public`, `protected` or `private`) of members in an aggregation is also determined by the type of the aggregation, `struct` or `class`.

Contexts. More generally, each scoping unit: file, function, `class`, `struct` or `namespace`, acquires a context from its enclosing scoping unit, may override this context in part or in whole, and passes on the modified context to elements enclosed in it.

In summary, as is the case in life, things must be put in context. Observe that in the last two items above, an enclosed element not only acquires properties of the enclosing one, but may also change them. A `protected:` statement in a `struct` changes the visibility attribute of its enclosing `struct`. We deal with this delicate issue later.

3 Paths of Acquisition

In our motivating examples, features were acquired through aggregation links. A natural question that arises is that of acquisition along other kinds of links among objects. In the MVC model [26] for example, it would be far-fetched to claim that the view is *part* of the model, but it would be very natural for it to *acquire* along the "views" link those aspects of the model that are relevant to displaying the model. The same can be argued for the OBSERVER design pattern [16] which can be thought of as a generalization of MVC. There are other design patterns such as PROXY and STATE in which acquisition might prove useful even across non-aggregation links.

3.1 Aggregation links

Acquisition along aggregation links is particularly interesting because of several properties that the containment relationship exhibits:

(i) an object may be a component of (directly contained in) at most one composite at any one time;

(ii) no object may be contained, directly or indirectly, in itself; and

(iii) all objects may be part of a containment hierarchy, i.e., all objects have aggregation links.

Property (i) eliminates the need to specify the link through which the acquisition is done. Property (ii) resolves the problem of circularity in acquisition. Property (iii) enables acquisition for all objects.

The forest topology of containment engenders an analogy between single inheritance and acquisition. This analogy even suggests "*environmental inheritance*" as an alternate term to acquisition, where the notion of *environment* refers to a list of all enclosing composites of a component.

Three more properties complement our understanding of the containment relationship:

(iv) composites export the ability to access the components they enclose as autonomous objects;

(v) the protocol of a composite does not depend on knowledge of its components; and

(vi) the protocol of a component does not depend on knowledge of its composite.

Together, these last three properties constrain the coupling between the composite and the component. Although property (iv) is sometimes used for separation of containment (Car-Door) from attribution (Car-Colour), it is not a pre-requisite for the two others, nor for acquisition. The protocol of objects exists even if they are inaccessible from outside the containment, and acquisition might be useful for such objects as well. [7]

Containment relationship has yet another property:

(vii) the "contains" relationship is covariant.

Specifically, if a class C_2 has a slot of class C_1, then the type of this slot in a subclass of C_2 is C_1, or C_1', a subclass of C_1. With the natural abstract superclass rule [21], and the proviso that insertion of components into slots can only be done in concrete container classes, we have that covariance of slot type is not only natural, but can also be checked statically. This form of covariance should be contrasted with covariance of function arguments, which is natural but unsafe.

Although general purpose programming languages reflect the situation in which "... the difference between *whole-part associations* (WPAs) and other associations is often only cosmetic and diagrammatic." [9], there are numerous application domains in which containment is essential and natural. In these domains, which include solid modelers, hierarchical databases, text structuring systems such as SGML, parsers and other language processors, acquisition should be done along aggregation links.

In order to restrict acquisition to aggregation links, we must be able to distinguish these from all other links. However, in examining programming languages we find that such a lingual distinction tends to be the exception rather than the rule: In reference-semantics languages such as SMALLTALK, there is no clear distinction between containment and other kinds of associations; in value-semantics languages such as LISP, all associations are containment in a sense.

The distinction is crisper in mixed semantics languages such as C++ and EIFFEL [32]: an object is interpreted to be contained in another if its value is stored in it (EIFFEL's **expanded**), while reference representation is used for non-containment associations. But despite the explicit claims [24] of the designers of BETA [31], reference semantics is not exclusive to associations in that language as in others. Reference semantics is used in many cases for implementation convenience, for overcoming problems of creating and managing large wholes and wholes with a variable number of parts, and for handling cases where garbage collecting environments prevent the realization of property (iv) with value semantics of the containment. Note also, that despite its smoothness and safety, property (vii) is not supported directly in mainstream languages.

The consideration of issues of acquisition may help in the dilemma of discerning WPA from other kinds of associations. This dilemma is expressed in Civello's words [9]: "While it is generally acknowledged that WPAs bind classes more strongly than other associations, there are no further rules or constraints to guide design and implementation decisions." If acquisition occurs along specific links then these links are more likely to be classified as containments, although as we have seen, there are cases of acquisition along other kinds of links.

The term *whole* in the acronym WPA does not coincide entirely with our understanding of the composite notion. In [9], Civello also suggests classification of wholes as *assemblies* in which WPAs are *functional*, or as either *aggregates* or *tuples* in which both WPAs are *non-functional*. We believe that acquisition along aggregation links should occur only along functional WPAs. However, it should also occur along *spatial or temporal inclusions* which are not WPAs according to Civello's taxonomy.

3.2 Nonaggregation links

Although our chief example for acquisition is through containment links, there are cases of acquisition of features along other links. The environment of an object might contain the sender of the message, the set of containers, or the creator of the object ([22] for example presents a language mechanism for allowing an object to inherit behaviour from its creator).

We generalize the concept of acquisition by allowing acquisition to occur through an arbitrary system of links that has a forest topology and is similar in structure to containment. By this we mean chiefly properties (i)–(iii) and (vii). Properties (iv)–(vi) are significant as well, but they are usually only a concern in containment in which the binding between the objects is so strong that a clear boundary must be set between them. Examples for containment-like hierarchies are the relation between an object and its creator [22] and ownership as in Car-Owner.

Another possible generalization is *multiple environmental acquisition*: this occurs e.g., in the armed forces and other large organizations where there are two chains of commands: professional and organizational. An artillery officer might report organizationally to the

[7] Conversely, we can remark that it is not even essential that the composition root itself be accessible. In a parse tree application, for example, there may be no need for direct access to the single composition root.

brigadier and professionally to the chief artillery commander. We leave this generalization beyond the scope of this paper, and, sufficing ourselves with the intuition and the motivation built upon the motivating examples, concentrate on issues of the theory of acquisition itself.

The main virtue of containments and containment-like hierarchies is the ability to use transitive closure in acquisition. If an object a is contained in b and b is contained in c, then a may acquire features from c, even if b does not acquire them. Current applications using attribute grammars are cumbersome because copy rules must be used to propagate the values of attributes. Great simplifications are achieved using what we may call *leap acquisition*, arising from this transitive closure.

Yet another generalization step is that of allowing acquisition across arbitrary links. However, with this generalization, the uniformity of the links is lost; the notion of transitive closures may thus lose its meaning. Although it may be technically possible to extend the definitions to enable leap acquisition in such a case, we will refrain from doing so. To smoothen the discourse, we limit the technical discussion to containment relationships only. It should be obvious that no generality is lost.

4 Static acquisition

Acquisition could be implemented by letting all objects store a reference to their immediate enclosing composite, if one exists. The standard binding of messages to methods can then be altered so that if a message is not recognized by a receiver, it is resent to its composite. Resends could be done in a *forwarding* manner, i.e., method execution in the context of the composite, or in a *delegation* manner, that is method execution in the context of the original receiver. [8]

The implementation of this seemingly-simple scheme is difficult in statically-typed compiled languages such as C++; sophisticated tricks such as the one presented in [11, Section 9.2] are required for tampering with the builtin dynamic binding mechanism. Implementation is more feasible, however, in SMALLTALK and other dynamically typed languages which poses runtime reflective capabilities. Nevertheless, in both implementations much is left to be desired in terms of safety and ease of use. The sender of a message must be familiar with the runtime environment of the receiver in order to know if the messages will be recognized or not. Lack of safety is also the source of difficulty in an implementation in C++-like languages: the protocol of an object can only be determined at runtime.

To obtain type safety, a programming environment supporting acquisition must be able to generate and prove predicates such as "instances of C_1 can only occur as direct components of instances of C_2", "instances of C_1 can only occur as direct or indirect components of instances of C_2", and "instances of C_1 may occur as direct components of instances of C_2". Further, since as explained above, the interface of a component lists the acquired features that are part of its interface, the environment must be able to make deductions of similar nature with regard to individual features.

4.1 Environmental polymorphism

The main difficulty in carrying out static analysis is the accounting for the environmental polymorphism, i.e., the uncertainty that stems from the multitude of potential configurations of the composition hierarchy at runtime. There are two sources to this uncertainty:

4.1.1 Containment freedom

The same component may belong to composites of different classes or even occur standalone. Examining Figure 5 for example, we see that Door may acquire *colour* and *airline*. A Door may be part of a Car or a part of an Airplane but never both. Acquisition of neither can be guaranteed.

4.1.2 Subtype polymorphism

Determining all possible composites a class may be part of is done by a traversal of the graph of classes and the composition links that connect them. Curiously enough, subtype polymorphism makes it necessary to examine, for each class encountered in the traversal, all of its superclasses, all of its subclasses and even all superclasses of all of its subclasses. To understand this,

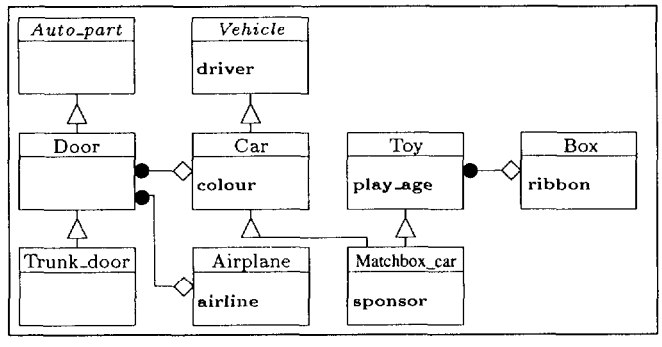

Figure 5: Environmental polymorphism

[8] The following section explains why forwarding is preferred over delegation for acquisition, but this distinction is of secondary importance here.

consider the two distinct ways in which subtype polymorphism may effect acquisition:

Component Polymorphism If a superclass of a certain class may be contained in a composite, then so might the class itself. In Figure 5, a Trunk_door may be contained in a Car.

Composite Polymorphism If a class may serve as a composite of a certain component, then all classes inheriting from this class are potential composites of that component. Consequently, when we go from the component to the composite through a specific composition link, there is uncertainty with regard to the actual class of the composite. If that composite is in turn a component of yet another composite, then its polymorphic nature must be taken into account.

In other words, composite polymorphism means that if a subclass of a certain class may be contained in some other class, then (non-immediate) instances of this class may also be contained in this composite. In Figure 5, we have that some Auto_parts may be contained in a Car.

More generally, in a multiple inheritance setting, non-immediate instances of a class that shares a heir with another class, may be contained in any composite that contains instances of that other class. In Figure 5, some Cars may be contained in a Box.

To handle cases where a component cannot be guaranteed to acquire a feature, we employ two standard techniques: a default action or value for missing features and a *guarded* computation mechanism similar to ML's **case** operator [36] and C's ternary operator, *cond* ? *exp1* : *exp2*, where *exp1* and *exp2* (the true and the false branches) are of the same type. These techniques should be used to capture uncertainty due to containment freedom and due to component polymorphism; protection against uncertainty due to composite polymorphism is more appropriately taken care of by dynamic binding. In the example we have that an Auto_part may have a *play_age* feature in its environment if it happens to also be an instance of Door which happens to also be part of an instance of a Matchbox_car. However, this property is not allowed to be acquired since checking whether or not it exists is nothing else than a baroque mechanism of runtime type information of objects—a technique that is better avoided whenever possible.

4.2 Kinships between classes

This subsection gives a precise meaning to the sentence "a class C_1 *may* be contained in a class C_2", written as $C_1 \sqsubseteq C_2$. This meaning accounts for containment freedom and for component polymorphism but specifically excludes composite polymorphism from the semantics of the word "may". In Figure 5, we will have that Door $\sqsubseteq$ Car, but that none of the pairs (Auto_part,Car), (Door,Vehicle), and (Door,Box) stand in the '$\sqsubseteq$' relationship.

Note that if $C_1 \sqsubseteq C_2$, then all features that C_1 acquires from C_2 must have default values or be used only with guarded expressions. Guaranteed acquisition of a feature can only be done if is known that "C_1 *must* be contained in C_2". The precise meaning for this sentence (Definition 4.3) includes all sources of runtime uncertainty, since in this case, a feature that C_1 acquires from C_2 can be used without guards in all possible configurations of the runtime hierarchy.

The remainder of this section gives formal definitions for the so-inclined readers. These definitions can be readily translated to algorithms. The following notations are pertinent: For two classes C, C', we write $C < C'$ if C inherits directly or indirectly from C' and $C \leq C'$ if $C < C'$ or $C = C'$. A class C has a *set of slots* $S(C)$. Members of $S(C)$ are pairs of the form $\langle n, D\rangle$, n being a named place holder for a component of class D. Also, let $S'(C)$ be the *set of all* inherited *slots* of C, i.e., $S'(C) = \bigcup_{C \leq C'} S(C')$.

For simplicity, we assume that no overloading occurs. Every slot name n is introduced in exactly one class which we denote by $Intro(n)$. That is to say, no name n appears in more than one set $S(C)$. (Overloading fans who dislike this restriction may use classes to tag overloaded names.) One exception to account for property (vii) applies, if $\langle n, D\rangle \in S(C)$ and if $C < C'$, $D < D'$, then we may also find that $\langle n, D'\rangle \in S(C')$. Still, $S'(C)$ would contain only one copy of n as part of the pair $\langle n, D\rangle$ (usual overriding).

W.l.o.g., there is also a unique frozen[9] *root* class[10] R which makes roots of composition trees; only instances of R can serve as such roots; there is a single slot $\langle r, R'\rangle \in S(\mathrm{R})$; only subclasses of R' can instantiate immediate components of roots.

4.2.1 May kinship

As explained above, when analyzing potential containment relations, we must look at classes that are related by a common subclass. If $D \leq C_1$ and also $D \leq C_2$, then an object contained in a slot defined in C_1 may acquire features defined in C_2.

[9] Frozen classes are classes that cannot be further used for inheritance.

[10] The root class R should not be confused with "Any" that is sometimes used to denote the root of the inheritance tree.

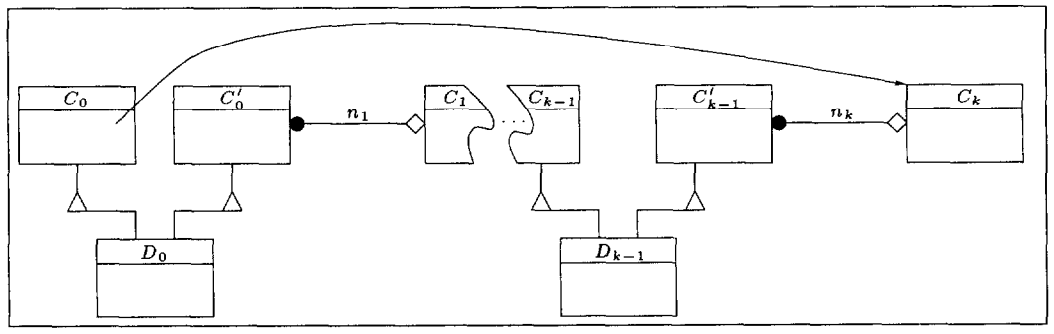

Figure 6: A containment path from C_0 to C_k.

We write $C \overset{D}{\multimap\!\!\bowtie\!\!\multimap} C'$ to denote that C and C' are related via a common subclass D. We write $C' \overset{n}{\multimap\!\!\diamond} C$ to denote that n is a slot satisfying $\langle n, C' \rangle \in S(C)$.

Definition 4.1 *A* containment path *is a sequence*

$$\pi = C_0 \overset{D_0}{\multimap\!\!\bowtie\!\!\multimap} C'_0 \overset{n_1}{\multimap\!\!\diamond} C_1 \overset{D_1}{\multimap\!\!\bowtie\!\!\multimap} \cdots \overset{D_{k-1}}{\multimap\!\!\bowtie\!\!\multimap} C'_{k-1} \overset{n_k}{\multimap\!\!\diamond} C_k$$

where C_k is the root class R. (Figure 6)

- *We say that a class D* dominates *a containment path π if there exists a class D_i on π, such that $D_i \leq D$.*
- *We say that a containment path π* covers *a class D if $D \leq C_i$ and $D \leq C'_i$, for a pair of classes C_i, C'_i on π.*

Dominance of a containment path starting with a class C_0 is a necessary condition for environmentally affecting C_0's interface. If D dominates a particular containment path (that starts with C_0), then C_0 is *potentially* contained in D. If *all* containment paths that start with C_0 are dominated by D, then C_0 is *necessarily* contained in D. May- and must-kinships are based on potential- and necessary-containment, respectively, with few additional constraints.

The concept of coverage captures uncertainty due to composite polymorphism, that is, the freedom of choosing the actual composites $D_0, D_1, \ldots, D_{k-1}$. A definition of may kinship has to ensure that nothing is revealed about the subtype of the composite. Suppose that we have that a class C "may be contained" in a class D. Then, the environmental acquisition mechanism makes it possible to determine in runtime for any specific object of C if it is contained in an object of D. We would like to limit this runtime power to exactly this. Specifically, if $D' < D$ or $D < D'$ then the environmental acquisition mechanism should not give rise to a possibility of determining whether an object of C is in an object of D or D'.

There are several other subtleties in the definition of may kinship which are not discussed in this proceedings version of this paper. As it turns out the appropriate definition for may kinship is:

Definition 4.2 *We say that a class C* may be contained *in a class D, $C \sqsubseteq D$, if all containment paths that start with C and cover D are also dominated by D.*

4.2.2 Must kinship

It is harmless to ignore some potential containment paths in may kinship. Not so for the must kinship. We need to examine *all* potential paths to ensure that an instance of the component can *only* be contained in an instance of the composite. Yet, must-kinship overrules some containments that satisfy this condition but their safety is coincidental in nature rather than captured in the design.

Again, we can obtain that an adequate definition for must kinship is:

Definition 4.3 *We say that a class C* must be contained *in a class D, $C \prec D$, if there exist a slot n such that all containment paths that start with C are dominated by D, pass through n, and $D = Intro(n)$.*

With regards to inheritance, the must kinship takes into account both component and composite subtype polymorphism as follows: $C \prec D$ only if $C' \prec D$ for any C' such that $C' \leq C$ or $C \leq C'$.

5 Acquisition of features

Now that the may and must kinships are elucidated, we can deepen our study of acquisition by investigating the issues arising when specific features are acquired.

5.1 Acquisition of routines

Inheritance, as a relation among classes, pertains only to features which are shared by all instances of the class: methods, structure and potentially also class variables. As explained in the introduction, acquisition, as a relation between classes, deals also with features which are particular to an object (attribute values). To gradually reach understanding of this, our study begins with examination of routines. Routines are conceptually simpler than attributes in the sense that they are read-only features. In most object-oriented systems (excepting perhaps Delphi) objects cannot replace routines defined in the class.

The simplest case of routine acquisition occurs when a class must be contained in another class. In this case, we can use the following EIFFEL-like syntax to declare an acquired routine:

```
class Paragraph
feature
acquire emphasize()
...
end Paragraph
```

(Recall that acquired features must be declared explicitly. After all, we do not want a Door to accidently acquire a method to start the engine.) The compiler must now check that there exists a class that defines *emphasize* such that Paragraph must be contained in that class. For example, it may find that an *emphasize* method is defined in a class Document and that Paragraph $\prec$ Document.

What happens if there exists some other class, say Section, that defines *emphasize*, and Paragraph must (or even just may) be contained in it? An instance of Paragraph would then acquire *emphasize* of the inner most composite object that has it, be it of class Document or Section. This is fine as long as Document.*emphasize* and Section.*emphasize* are "basically the same" feature. That is to say, if there is a common ancestor of Section and Document, say Formatting_Entity that defines *emphasize*.

If there is no such ancestor, then Document.*emphasize* and Section.*emphasize* are tied only by means of name *overloading*. Such a tie is coincidental in nature; there is no certainty of agreement or even conformance between the types of the two features. If the features were intrinsically related, then a skilled designer should have captured this in a common ancestor.

We allow acquisition of an overloaded attribute only if the overloading ambiguity is resolved with explicit qualification of the defining class name. Incidentally, another consequence of the above deliberation is that a good acquisition system must support multiple inheritance so as to enable factoring of common features into appropriate base classes.

The next case to consider is acquisition in may kinship. Consider two classes C_1, C_2 such that C_1 may be in C_2, but it is not guaranteed that C_1 must be in C_2, i.e., $C_1 \sqsubseteq C_2$ and $C_1 \not\prec C_2$. Suppose that C_2 defines a feature f. It would be unsafe to invoke, by means of acquisition, the feature f from C_2 unless f is defined also in C_1. If $C_1.f$ takes precedence over $C_2.f$, then $C_2.f$ can never be called, and this would render acquisition pointless.

Conversely, an interesting and useful semantics is obtained when letting $C_2.f$ take precedence over $C_1.f$. For example, a black Door put in a white Car should turn white. But once removed from it, it should show black again. Notice that the precedence is in opposite direction than what happens with ordinary inheritance. A class that inherits a feature may override it. A class that acquires a feature must supply a *default* implementation to it; the acquired feature takes precedence over that default.

Routines that are acquired through a may kinship are defined as follows:

```
class Window
feature
may acquire help() default is
...
end help
...
end Window
```

In inheritance a routine may refine its inherited implementation. Is it also possible for a routine to refine its acquired implementation? The default implementation of *help* cannot call its acquired version, since it is invoked exactly when there is not such an acquired version. In contrast, in the case of must kinship, refinement is possible.

```
class Quotation
feature
acquire left_margin(): Length overrule is
return acquired left_margin() + 1cm
end left_margin
...
end Quotation
```

The **overrule** keyword signifies that the implementation of *left_margin* in the current instance of Quotation takes precedence over the acquired one.

A **default** implementation in a must kinship is meaningless. All that remains is therefore to consider the

case of an **overrule** implementation in may kinship, which makes sense only if the overruling implementation calls the acquired one. Although it is not entirely clear that this must be supported, we describe how it can be done. For a type safe implementation, we are obliged to introduce a language construct mirroring ML's **case**, where it is used for safe examination of variant records.

```
class Enumeration
feature
may acquire item_sep(): Length overrule is
return 0.8 * acquired item_sep() guard
0.5cm
end item_sep
...
end Enumeration
```

The **guard** clause provides a default value to the **return** expression when the **acquired** feature *item_sep* is not present. In words, the above states that item separation in a nested enumeration is 80% of the innermost enclosing enumeration and that it is half a centimeter in an external enumeration. Note that Enumeration may probably acquire from itself.

The **guard** clause makes it possible to call a **may acquire** routine from outside of the acquiring class. Therefore, it is deprecated but not forbidden to declare such a routine with no **default** implementation:

```
class Component
feature
may acquire foo(): Boolean;
-- No body for foo here!
...
end Component
...
x: Component
...
if x.foo guard true then ...
```

Table 1 summarizes the proposed syntax and semantics of routines acquisition.

5.2 Context of execution of acquired routines

So far, we tacitly ignored the issue of the context of execution of acquired routines. There are two possible semantics corresponding to ordinary (static binding) and `virtual` (dynamic binding) of function members in C++. This distinction is also known as, especially in the context of links between distinct objects, the quarrel between forwarding and delegation semantics.

Forwarding means that the context of the initiating object is forgotten during the execution of an acquired routine of this object. If this routine uses a certain feature, then this feature is understood in the context of the object of the *current* execution thread. Delegation means that the feature is sought in the context of the *initiating* object.

Dynamic binding and static binding generalize delegation and forwarding in prescribing the context of features selection also for polymorphic code that is external to the class. For example, a non-method routine that has a formal parameter of a certain class, may also receive an object of any of its subclasses. Dynamic binding dictates that the dynamic (actual) class of an object is used in interpreting the messages sent to it; in static binding, the static (declared) class is used instead.

Since acquisition does not impose subtyping, only the more restricted terminology is relevant. Put differently, a component cannot be used in a general context where a composite is expected. In the case of may kinship, such use is not type safe because it cannot be guaranteed that the component has or can acquire all the features the composite has. Even with must kinship, there is the problem that not all the features of the composite are sensible for the component–it is not appropriate to send a message to a door enquiring it for its number of wheels, even though it is guaranteed that a door is always a part of a car.

Let us therefore limit the discussion to the question of binding in a method, i.e., a routine defined within a class (as opposed to a general routine). Suppose that such a method of a class x uses a feature f on the current object. Then, in the forwarding semantics, all that is required in order to guarantee that f is found in run time, is to check that it is defined in x. A moment's reflection would show that this static check covers also delegation, even in the case of may kinship. The search for f is conducted along the chain of "contained in" links starting from the initiating object. The search must succeed since the chain includes also the current object of class x that defines f.

The fact that delegation semantics is type-safe does not necessarily imply that it should be preferred over forwarding. There are several good reasons to use forwarding semantics in all but very special cases. First, it is counter intuitive to have different semantics for internal and external routines. This would imply, for example, that the set of acquired features varies depending on whether you view it from inside or from outside the class. Second, acquisition encompasses a sophisticated mechanism of polymorphism by itself; delegation adds to this, and to the usual sub-type dynamic binding,

Kinship	Declaration		Syntax	Comments
may	not embodied		**may acquire** *f*	Use with **guard**
	embodied	default	**may acquire** *f* **default is** ...	Safe to use; no refinement
		overrule	**may acquire** *f* **overrule is** ...	Safe to use; refinement with **guard**
must	not embodied		**acquire** *f*	Safe to use.
	embodied	default	N/A	Senseless—never invoked
		overrule	**acquire** *f* **overrule is** ...	Safe to use; safe refinement

Table 1: Summary of routine acquisition.

yet another dimension of complexity. Third, in many cases where delegation semantics is required, it can be implemented by means of inheritance. For example, a method for computing the paragraph indentation level in a text processing system could read

parindent(): Length
return *left_margin*() + 1cm
end *parindent*

If *parindent* is an acquired feature, then we must compute *left_margin* within the context of the acquiring object. However, a better design would make *parindent* an inherited, not acquired feature. Fourth, within the semantics of "inherited" attributes in attribute grammars (one of the thoroughly studied examples of acquisition), the attributes are computed in a local context and only then propagated further.

There could be cases in which delegation semantics may be required also for acquisition. For example, one may want to apply different algorithms for computing the paragraph indentation in footnotes and in ordinary text. For that reason, we do allow delegation semantics for acquisition as well, and suggest that they both may play an important role. This is in contrast to the case of inheritance, in which the existence of non-`virtual` function members have no useful semantics, and their existence is just another symptom of the C++ design philosophy of not penalizing a programmer for an unused language feature.

5.3 Acquisition of attributes

Many of the conclusions drawn in the study of acquisition of routines are also applicable to acquisition of attributes. If we forget for a moment that values may be assigned to attributes, they can be treated as functions with no parameters in which no overriding is allowed.

The picture is complicated though if assignment is recalled. If an acquired attribute is also embodied in the acquiring class, then a-priori, the acquired value should take precedence. Later, the precedence could change. It should be possible to paint just the door green after it had been placed in a white car.

We propose the following operational model for the precedence of embodied attributes: With each such attribute we associate an auxiliary flag that controls whether the component takes precedence over the composite or vise versa. The flag may be in one of two states: **default** or **overrule**. The semantics of these two are in accordance with the **default** and **overrule** keywords used in defining bodies for routines.

Initially, when an object is created and the embodied attribute is initialized, the flag is **default**. When an attribute of a certain object is modified, its auxiliary flag becomes **overrule**. In addition, the flags of all embodied attributes acquiring the modified attribute become **default**. Thus, by painting the car blue, all doors become blue as well. By further painting a particular door of the same car green, the door's handle becomes green as the door, but the car as a whole remains blue.

When a free object is inserted into a composite, all auxiliary flags of that object are set to **default**, thus supporting the black-white door colour flipping behaviour described above. The flags do not need to change when an object is removed from a composite.

There are attributes which have a special *undefined* value. If an attribute is assigned *undefined*, then its flag becomes **default**. The auxiliary flag of the embodied attribute that acquire from this attribute does not change in this kind of assignment.

In a Delphi like system, in which routines can be overridden by single objects, similar semantics applies to routines.

Table 2 summarizes the proposed syntax and semantics of attribute acquisition. As it is for routines, the **acquire** clause for an attribute is either unmodified—signifying that acquired value is not guaranteed or accompanied with a **may** modifier—in which case no such guarantee can be made. In both cases, the attribute may be also **embodied** in the class. An embodied attribute may have an *initializer*, i.e., an expression used for initializing it at the time of object construction. Acquired attributes can be used safely, except in the case of **may acquire** with no **embodied**, where they should be used with **guard** (type safety Chauvinists

Kinship	Declaration	Syntax	Comments
may	not embodied	**may acquire** *f*	Use with **guard**
	embodied	**may acquire** *f* **embodied** [:= ...]	Safe to use; initializer may use **acquired** value with **guard**.
must	not embodied	**acquire** *f*	Safe to use.
	embodied	**acquire** *f* **embodied** [:= ...]	Safe to use; initializer may use **acquired** value safely.

Table 2: Summary of attribute acquisition

may want to forbid this case). The initializer may use the **acquired** value, but only with **guard** in the case of **may acquire**. In contrast with Table 1, **default** and **overrule** semantics is determined dynamically as described above and therefore is not described in this table.

The *rvalue* of an acquired attribute is one of the embodied attributes incident on the "contained in" chain that starts at the object. There are three possible semantics of the *lvalue* of such an attribute: the embodied attribute in the object itself, the nearest enclosing attribute, and the effective enclosing attribute.

We envisage a system in which assignment is normally done to the embodiment in the object but that has provisions for the other two semantics. Assignment to the nearest enclosing in the case of no embodiment is useful, e.g., for upward propagation of "synthesized attributes" in an attribute grammar application. For example, inside the parse tree of a nested C++ `class`, an assignment to an acquired symbol table should occur at the nearest enclosing scope that embodies a symbol table.

Assignment to the effective attribute is also useful. In a drawing program, it should be possible to repaint a distinctly painted part of a whole even through one of its subparts.

6 Comparison to other models and related works

In this section we briefly compare acquisition to other related work. We first explain why some common OO constructs do not constitute by themselves an acquisition system and then turn into comparing our work with recent advances.

6.1 State as it is recorded in instance variables

It is possible to implement acquisition using object state. An object may contain a pointer to its composite (immediate container). By following the chain of pointers it is possible to implement all of the intricacies of the acquisition. Indeed, systems such as TEX, InterViews, TurboVision and many attribute grammar compilers do exactly this kind of emulation. One may therefore argue that the state of an object, including the value of this pointer includes all information required for acquisition. We disagree with this claim because of the following two reasons:

First, as should be clear by now, acquisition is far from being trivial. Its inherent complexity cannot be covered up. Re-implementation, as done for example in the software systems mentioned, is bound to be complicated, inefficient, and in many cases lacking a well defined semantics. It is instructive to consider the case of attribute grammars. Although there is a huge body of work on their efficient implementation, they have failed to become popular. We feel that this is partly due to the fact that the algorithms were not packaged in a well defined language construct, and the need to enumerate all "inherited" attributes puts a heavy burden on the user.

Second, we feel it is wrong to blur the boundary between objects by stretching the meaning of the term "state" to include external objects to which pointers are stored. A network of objects, including perhaps all objects of a system, could be thus made into a single object. Further, since in general pointers are not bidirectional, and since there might be variables pointing to various locations in the network, a complicated semantics would have to be called in to describe the complex relationship between a subobject which corresponds to connected component of a directed graph. Instead, we feel that if there is a consistent pattern of acquisition of properties from one object by another, then it is better to define an abstraction mechanism that captures this pattern. Acquisition, and in particular the overrule/default mechanism described above, gives a good balance between state change, the change undertaken by the object itself, and the environmental influence.

6.2 Inheritance

Inheritance is a relation between classes, whereas acquisition pertains to individual objects. Other reasons why inheritance cannot serve for acquisition are given in the following discussion of dynamic inheritance.

6.3 Dynamic inheritance

Here we take this term in its restricted sense—a sense also called "configurable inheritance" [18]: if a class C is declared as inheriting from C', then the sub-object of objects of C, which correspond to the base-class C', may also be of a class C'' which also inherits from C'. Loosely, C may also inherit at run time from any class C'' which inherits from C. Stroustrup [42, Section 12.7] reports on a C++ extension proposal accompanied by an implementation experiment to that effect. Had this proposal of "delegation", as it is called there, been accepted, it could have only approximate acquisition, falling short in several important ways:

1. Dynamic inheritance, just as a static one, induces a subtype relationship. This is usually undesirable in acquisition: a component is not necessarily a subtype of its enclosing composite. Observe that `private` inheritance merely restricts the visibility of sub-typing (as suggested in [40]) but does not exclude it altogether.

2. There is usually an orthogonal hierarchy classifying the kinds of objects that may occur in the containment hierarchy. Mixing the two hierarchies at the cost of the complexities of multiple inheritance and at the risk of blurring the distinction between the two hierarchies is worse.

3. It is inherent to acquisition that an "inherited sub-object" (a composite) is shared by many objects (its inheriting components). Such sharing contradicts our usual understanding of inheritance.

This latter hindrance is the main reason for the failure of this experiment.

6.4 Prototypes

To emulate an acquisition in a prototyping system one would, as another approximation, replace substitute containment links with delegating ones.

Several problems arise. The first, and perhaps the easiest, is that of delegation vs. forwarding. As mentioned above, both semantics are feasible and useful for acquisition. Clearly, adding support for forwarding to runtime systems, such as that of SELF, which already support delegation should be a simple task.

Another problem is that, the containment links should be distinguishable from other links. This again could be done via relatively easy syntactical changes. Another change required is reversing the links directions so that the composite has slots for the components and not vice-versa. Although no delegation system that we know of implements this, such an addition does not seem too problematic.

The third, and most difficult problem: delegation systems, almost by nature, subvert strong typing. Beyond the usual benefits of strong typing, such as improved efficiency and reliability, we argued above that acquisition should be done in a strongly-typed manner. In this respect, our research can be viewed as an attempt to investigate strong typing of a restricted form of a delegation system. The more general problem should be the subject of another investigation.

6.5 Other related work

Blake and Cook [2] deal with the issue of support to containment hierarchy in OO languages. They compare part hierarchies with inheritance and delegation and provide a mini-taxonomy for kinds of containment relationships. They also identify the dilemma of the visibility of parts, and argue strongly that exposing the parts does not violate encapsulation. (Thus supporting property (iv) of ours). Specifically they propose, implement and report on the lessons learned from a SMALLTALK extension in which the composite forwards messages to its components. Another aspect in which their work is different from ours is that they tacitly ignore the question of strong typing and optional components. In our model strong typing and optional "parents" play an important role. We were unable to determine if Blake and Cook's work makes the distinction between part and attribute links.

Kim et. al [23] examine the question of composite objects in OO database systems. They also argue that the composition links should be distinguishable from other links. They provide a formal definition of the data model of composite objects and show how they might be used in a database system by addressing questions such as locking and efficiency. One of their interesting contributions is the support for versions of composite objects. (This is similar to the work in Vesta [6].) As it is common in databases, it is assumed in their work that parts have external visibility. No treatise is given to the question of propagating properties inside the composite object. We consider using their results in testing our proposal on large inputs, such as legacy systems, which

need to be reverse engineered.

Kristensen [27] argues for and proposes lingual mechanisms to support general complex associations, including special notations for containment. In this sense and in the strong ties his work has with formal methodologies of analysis and design[11], it is more general than ours. However, acquisition is much more powerful than a mere existence of a containment link.

7 Conclusions, Open Questions and Further Research

This paper demonstrated the need for down propagation of attributes in containment hierarchies. This is just a special case of a phenomena that often shows up in the development of even modest-sized OO systems: an object behaviour depends on its surrounding environment. The broader question that we try to solve is the modeling of environmental affect, i.e., the ways in which the environment participates in setting the properties of an object.

It is clear that an object is not a specialization of its container or any other element of its environment. Therefore, it is not appropriate to let an object *inherit* from its environment. Instead, the traditional technique for modeling such an affect is by interfaces that are tailored for each kind of environmental link. Despite its appealing simplicity, this approach suffers from several drawbacks:

1. Instead of the convenient separation between public interface and implementation, objects must posses dedicated environment interface(s).
2. The design and implementation of such interfaces is difficult and repetitive.
3. Environmental affect by non-immediate neighbours is tricky. What we called leap acquisition is a powerful tool that is difficult to use in the absence of a good understanding of the involved issues.

The new acquisition mechanism we advert offers a compromise between inheritance and a tailored solution. It is not as elegant and easy to understand as inheritance (but then, so are real life programming problems) but it is much more appropriate for modeling environmental affect. On the other hand, acquisition is more orderly and more elegant than a tailored dynamic solution which tends to be error-prone, inefficient and discouraging to the implementor.

[11] In this respect, it should be compared to the work of Civello [9]

Our investigation revolved around the example of acquisition across aggregation links. The results however, are applicable directly to acquisition in other forest hierarchies and from immediate general neighbours. The generalization to other network topologies and leap acquisition across non uniform links is a subject for further research.

Among the surprising discoveries of this research we include the illusive character of the definitions of the may and must kinships: We saw that in order to prove that an object of a certain class must be contained in an object of another class, one has to consider carefully the containment relationships between all super- *and* subclasses of both these classes. Also, the may kinship may, if not defined properly, capture instead of structure freedom, the dynamic type of objects; information that is better handled by other means in good object-oriented programming.

Most design methodologies represent relationships and objects, but offer almost no rules or guidelines of how these should interact with inheritance. A promising research objective is better insight of the interdependencies between inheritance and containment and other links among objects. This hopefully will result in a coherent set of rules similar to property (vii) to guide design and enable more structured reasoning.

Even though our understanding of the "permissible" topologies in the inheritance-containment diagram is lacking, and we allow many constellations which should never occur in a well designed system, static type analysis can eliminate many possibilities and help a dynamic implementation. We can show that this analysis can be done in $O(1)$ calls to a directed graph s-t connectivity algorithm.

We identified the basic rules of acquisition of routines, and in particular how refinement and overriding carry on to it. The term "*environmental polymorphism*" was coined to designate polymorphic behaviour of objects due to their respective environments. We proposed an operational model for environmental polymorphism that uses a system of defaults and overrulings. The model seem to capture the needs of current applications.

With regard to implementation complexity, it is easy to imagine an efficient implementation if the containment trees are static: if each acquisition is realized using a pointer then the time to access an acquired feature is $O(1)$. The problem becomes more difficult if we want to cater for tree updates: insertion and deletion of components.

Two solutions come to mind: In the first, acquisition pointers are updated with each tree operation. This is an $\Omega(n)$ (worst case) time operation, where n is the

number of objects in our system. In the second, there are no acquisition pointers; instead each object has a pointer to its container. In this solution, updated are efficient, running in $O(1)$ time, but queries are $\Omega(n)$. In both solutions though, the setting to **default** of an auxiliary flag in a whole sub-tree as a result of an assignment to an attribute is an $\Omega(n)$ time operation.

There is a sophisticated implementation approach that gives $O(\lg n)$ amortized (and at the cost of greater programming complexity, even $O(\lg n)$ worst-case) for all operations: insertions, deletions, query and flag flipping. This is done using data structures based on Tarjan and Sleator's dynamic trees [39] (See [44] for a text book exposition.) Unfortunately, this can only be done if there is only one kind of an acquired attribute in the tree. If there are m kinds of attributes in the system, then the insertions and deletions time increase by an m factor. What's worse, the storage requirements become $O(mn)$. Our data structures intuition leads us to believe that much better implemenations are possible, but their existence remains an open problem.

This research of course only opens the road to the understanding and using acquisition. There are many lessons to be learned and data to be gathered from a large scale exploitation of this new abstraction mechanism.

Acknowledgment We thank William Cook for his thoughtful remarks on an early version of this paper.

References

[1] Accent Software International, Jerusalem, Israel. *Dagesh User's Guide, Israel's National Word Processor for Windows*, 1994.

[2] E. Blake and S. Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In *European Conference on Object-Oriented Programming*, number 276 in LNCS, pages 41–50. Springer-Verlag, 1987.

[3] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[4] Borland International, Scotts Valley, CA. *Pascal Turbo Vision Programming Guide*, 1992.

[5] A. H. Borning. Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference*, pages 36–40, Dallas, TX, 1986.

[6] M. R. Brown and J. R. Ellis. Bridges: Tools to extend the vesta configuration management sys tem. Technical Report 108, digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, June 1993.

[7] M. Bryan. *SGML an Author's Guide to the Standard Generalized Markup Language.* Addison-Wesley, 1992.

[8] L. Cardelli and P. Wegner. On understanding types, data abstractions, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[9] F. Civello. Roles for composite objects in object-oriented analysis and design. In *Proceedings of the 8^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 376–393, Washington, DC, USA, Sept. 26 - Oct. 1 1993. OOPSLA'93, Acm SIGPLAN Notices 28(10) Oct. 1993.

[10] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice-Hall, 1991.

[11] J. Coplien. *Advanced C++ Programmings Styles and Idioms.* Addison-Wesley, 1992.

[12] B. J. Cox. *Object-Oriented Programming - An Evolutionary Approach.* Addison-Wesley, 1986.

[13] S. E. Fahlman. NETL: A system for representing and using real-world knowledge, 1979.

[14] J. D. Foley and A. V. Dam. *Fundamental of Interactive Computer Graphics.* Addison-Wesley, 1984.

[15] I. R. Forman, S. Danforth, and H. Madduri. Composition of before/after metaclasses in SOM. In *Proceedings of the 9^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [35], pages 427–439.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, 1995.

[17] J. Gil and D. H. Lorenz. SOOP – a synthesizer of an object-oriented parser. In *Proceedings of the 16^{th} International Conference on Technology of Object-Oriented Languages and Systems*, pages 81–96, Versailles, France, Mar. 6-10 1995. TOOLS 16 Europe Conference, Prentice-Hall.

[18] J. Gil and R. Szmit. Software boards via configurable objects. In *Proceedings of the 14^{th} International Conference on Technology of Object-Oriented Languages and Systems*, CA, Aug. 1994. TOOLS 14 USA Conference, Prentice-Hall.

[19] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[20] C. F. Goldfarb. *The SGML Handbook.* Clarendon Press, Oxford, 1990.

[21] W. L. Hürsch. Should superclasses be abstract? In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8^{th} European Conference on Object-Oriented Programming*, number 821 in LNCS, pages 12–31, Bologna, Italy, July 4-8 1994. ECOOP'94, Springer-Verlag.

[22] G. Kiczales. Traces (a cut at the "make isn't generic" problem). In S. Nishio and A. Yonezawa, editors, *Proceedings of the International Symposium on Object Technologies for Advanced Software*, number 742 in LNCS, pages 27–42, Kanazawa, Japan, Nov. 4-6 1993. First JSSST International Symposium, Springer Verlag.

[23] W. Kim, J. Banerjee, H. T. Chou, J. F. Garza, and D. Woelk. Composite object support in an object-oriented database system. In *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [34], pages 118–125.

[24] J. L. Knudsen, M. Löfgren, O. L. Madsen, and Magnusson. *Object-Oriented Environments, The MJØLNER Approach.* Object-Oriented Series. Prentice-Hall, 1993.

[25] D. E. Knuth. *The TEX Book*, volume A of *Computers & Typesetting.* Addison-Wesley, 1986.

[26] G. E. Krasner and S. T. Pope. A cookbook for using the model view controller user interface paradigm in SMALLTALK-80. *Journal of Object-Oriented Programming*, 1(3):26–49, Aug.-Sept. 1988.

[27] B. B. Kristensen. Complex associations: Abstractions in object-oriented modeling. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [35], pages 272–286.

[28] L. Lamport. LATEX: *A Document Preparation System.* Addison-Wesley, 1986.

[29] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [33], pages 214–223.

[30] M. A. Linton, J. M. Vlissides, and P. R. Calder. Interviews: A C++ graphical interface toolkit. Technical Report CSL-TR-88-358, Stanford University, Stanford, CA 94305-2192, July 1988.

[31] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison Wesley, 1993.

[32] B. Meyer. *EIFFEL: The Language.* Object-Oriented Series. Prentice-Hall, 1992.

[33] OOPSLA'86. *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, Sept. 29 - Oct. 2 1986. Acm SIGPLAN Notices 21(11) Nov. 1986.

[34] OOPSLA'87. *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Orlando, Florida, USA, Oct. 1987. Acm SIGPLAN Notices 22(12) Dec. 1987.

[35] OOPSLA'94. *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, Oct. 23-27 1994. Acm SIGPLAN Notices 29(10) Oct. 1994.

[36] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, 1991.

[37] Programmer's hierarchical interactive graphics system, 1986.

[38] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[39] D. Sleator and R. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 24, 1983.

[40] A. Snyder. Encapsullation and inheritance in object-oriented programming languages. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [33], pages 38–45.

[41] C. Strachey. Fundamental concepts inprogramming languages. In *Lecture Notes for the International Summer School in Computer Programming*, 1967. Copenhagen, Denmark.

[42] B. Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, Mar. 1994.

[43] Sun. *OpenWindows Library Manual.*

[44] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA, 1983.

[45] T. Thompson and N. Baran. The NeXT computer. *Byte*, 13(12):158–175, 1988.

[46] X. T. Trinh, J. T. van Tran, and C. Balkowski. Physics of nearby galaxies: Nature or nurture? In *Proceedings of the 27th Moriond Astrophysics Meetings*, Les Arcs, Savoie, France, Mar.15-22 1992. Editions Frontieres.

[47] D. Ungar and R. B. Smith. SELF: The power of simplicity. In *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [34], pages 227–241.