Encapsulation and Aggregation

Stuart Kent and Ian Maung*
Department of Computing
University of Brighton BN2 4GJ
England

Stuart.Kent@brighton.ac.uk, Ian.Maung@dcs.warwick.ac.uk

Abstract

A notion of object ownership is introduced as a solution to difficult problems of specifying and reasoning about complex linked structures and of modeling aggregates (composite objects). Syntax and semantics are provided for extending Eiffel with language support for object ownership annotation and checking. The ideas also apply to other OOPLs such as C++.

1. Overview

In section 2, we analyse the requirements of a mechanism for implementing the components of an aggregation or composite object, and we evaluate the extent to which the techniques of inheritance, expanded type attributes, reference type attributes and selective export satisfy these requirements. We restrict consideration to examples of composite objects from the solution (programming) domain. In section 3, we propose the use of reference type attributes with ownership properties to implement components and propose a syntax and semantics for object ownership. Section 4 shows how object ownership can be used in the natural modelling of aggregation structures identified in the problem domain. Section 5 discusses related work and section 6 concludes with a summary of contributions and issues for further investigation.

1.1 Terminology

We adopt the Eiffel terminology for OO systems [Meyer 88,92] throughout this article.

* Now with: Department of Computer Science, University of Warwick, Coventry CV4 7AL, England.

2. Implementing Components of an Aggregation

We study two important examples of composite object classes from the EiffelBase[1] (data structure) class library to motivate a list of requirements for a mechanism for implementing the components of a composite object.

2.1 Stacks with Array representation

Consider an implementation of the stack ADT using an array to represent the contents of the stack. A stack object is thus a composite object with a component array object. In Eiffel, there are at least three reasonable ways to implement this:

1. Define ARRAY_STACK[T] as an heir of ARRAY[T] and hide (make secret) all the exported features of ARRAY[T] from clients of ARRAY_STACK[T]. (see [Meyer 94] 3.10.4, p.111)
2. Define a secret attribute, rep : ARRAY[T] in ARRAY_STACK[T]. ([Meyer 88]p.118-119)
3. Define an attribute, rep : **expanded** ARRAY[T] in ARRAY_STACK[T] (see [Meyer 92]p.204-205).

If the last option is used and rep is not a secret attribute then clients of ARRAY_STACK can access its complete internal state, ignoring the LIFO property. This violates information hiding but not encapsulation, since assignment for expanded entities is field-by-field copy and thus a client can only tamper with a copy of the internal representation of the stack, not the representation itself.

The concepts of encapsulation and information hiding are of fundamental importance to this discussion, but unfortunately (as with much OO terminology) they do not have a universally accepted meaning. For our purposes, the following definitions will suffice:

- Encapsulation means that the components of a composite object cannot be modified except by feature calls to the composite object, or by calls from the composite object to its components. Encapsulation limits and controls aliasing and interference, thereby simplifying reasoning about and understanding object systems.
- Information hiding means that the components of a composite and their states cannot be accessed by clients. Information hiding limits and controls the dependence of clients on the supplier (composite object) internal representation, thereby localizing the effects of changing this representation during system maintenance.

In fact, the last option is not available in Eiffel since ARRAY has a creation procedure with an argument, violating a validity rule of Eiffel [Meyer 92] p. 284. This also overloads the meaning of expanded types, which (we believe) should be used to implement value types [Kent 95, Cook 94], such as the basic types INTEGER and BOOLEAN and other obvious value types such as PAIR[T1,T2] and other tuple types.

If the second option is used then it is possible for the encapsulation of ARRAY_STACK instances to be violated e.g. by defining an exported feature, *expose*, that returns a reference to the object attached to *rep*. Of course, it is fairly unlikely that the designer of ARRAY_STACK would provide such a feature. However, a distant descendant of ARRAY_STACK might introduce such a feature, without its designer realizing the implications. This means that assumptions that clients of ARRAY_STACK may be relying upon can be invalidated by descendant instances, when polymorphism is exploited.

[1]Note that we are not modelling container (data structure) objects as aggregates of the objects they contain. In the two examples given, a stack is an aggregate with an array component object, and a linked list is an aggregate with its linkable cells (but not its elements) as components.

If the first option is used, then inheritance has been used for code reuse and ARRAY entities should not be polymorphically attached to ARRAY_STACK entities (because ARRAY_STACKs hide some ARRAY features to prevent encapsulation violation and allow information hiding). This overloads the meaning of the inheritance relation, which in Eiffel, defines that a child should conform to its parents, and thus makes type-checking much more complicated [Meyer 92]Chapter 22.

If we require a number of alternative implementations of stacks e.g. stacks implemented using linked lists or arrays, then using inheritance we must introduce classes LINKED_STACK[T] and ARRAY_STACK[T] etc. inheriting their representation from LINKED_LIST[T] and ARRAY[T] respectively. Alternatively, we might try using a reference of deferred type, SEQUENCE[T], and we can use polymorphism to allow this attribute to be attached to an object of type LINKED_LIST[T] or ARRAY[T] at run-time. This is again not possible for the approach of using an attribute of expanded type, since expanded types cannot be deferred, and entities of expanded type are non-polymorphic. It is also likely that problem domain aggregations will have components of deferred or polymorphic type (see Section 4 for an example).

2.2 Linked list

For the purposes of this discussion, we assume the interface and implementation of LINKED_LIST from EiffelBase. A rationale for the interface design is given in [Meyer 88] Chapter 9, while implementation details are discussed in [Meyer 94] section 6.6.

A linked list (see Figure 1) can be modelled as an aggregation of its linkable elements. For linkable cells, neither the inheritance or expanded type attribute approach is possible, because LINKABLE is a recursive class (i.e. a client of itself). A LINKABLE object cannot inherit its *right* attribute (this would require LINKABLE to inherit from itself). An expanded class cannot have an attribute of its own type, as this would require its instances to be infinite objects. Hence LINKABLE cannot be an expanded class, although there can be instances of the expanded type, **expanded** LINKABLE[T]. Also note that if LINKABLE was expanded then, for example, insertion and deletion would require expensive deep copying and mean that the list was of fixed size, defeating much of the purpose of linked structures (efficient shallow reference copying and dynamically varying size).

Hence the linkable elements of a linked list must be instances of a reference type. Unfortunately, as mentioned in the previous subsection, making the linkable elements into secret attributes does not guarantee encapsulation or information hiding.

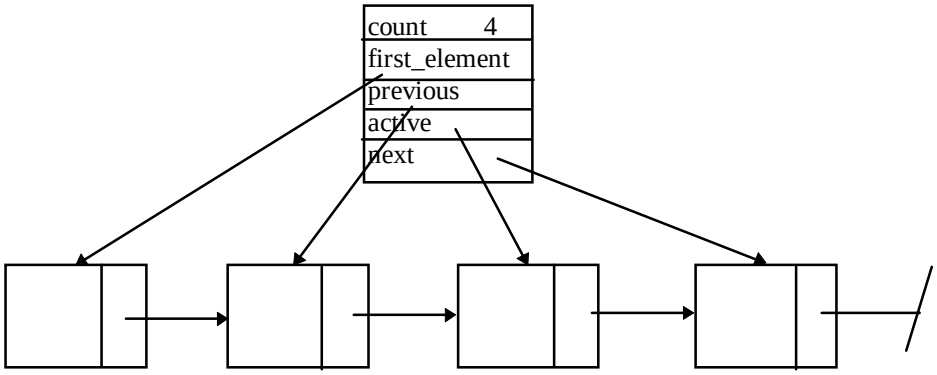

Figure 1 : A typical instance of LINKED_LIST.

2.2.1 The EiffelBase Solution : Selective Export

Although selective export works for LINKED_LISTs and LINKABLEs as implemented in EiffelBase, it does not provide the appropriate level of protection. Selective export provides protection at the class level, not the object level. For example the features *right* and *put_right* of linkables are exported only to the classes LINKABLE and LINKED_LIST. This means that no objects of other types can call these features and thus access or change the cell to the right of another cell.

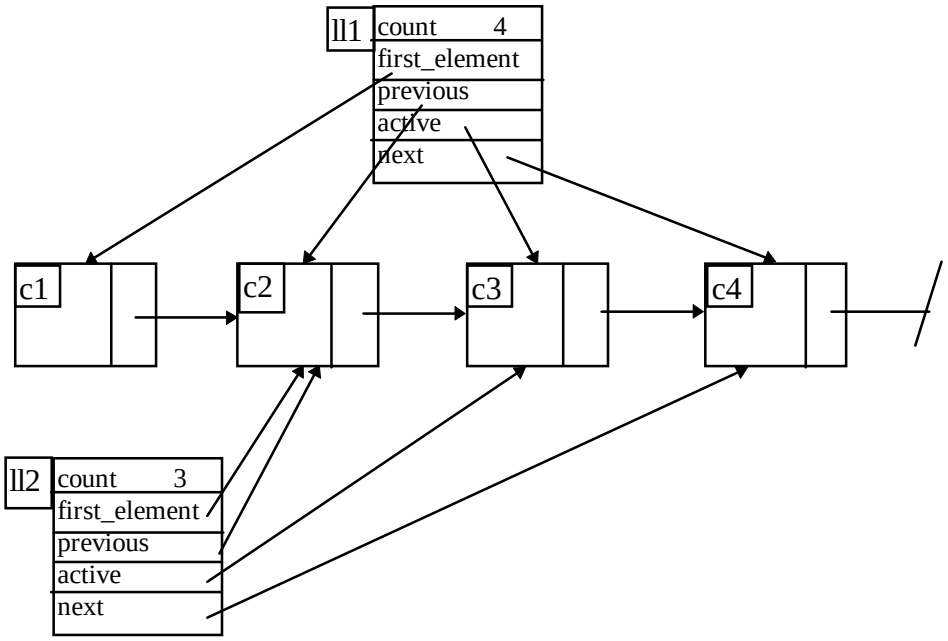

Figure 2 : Lists sharing linkable elements

One advantage of the selective export approach is that compliance to selective export restrictions is statically checkable.

The export restrictions still allow two distinct linked list objects to share the same linkable element and thereby interfere with each other. A simple example is shown in Figure 2, where, ll2 removes cell, c3, thereby changing the size of both ll1 and ll2. Now a client tries to access the fourth item in ll1, the **`require`** clause evaluates to true but a run-time error occurs. Essentially, an invariant of ll1 has been violated.

2.2.2 Class interface restriction

If we examine the interfaces of the classes LINKED_LIST and LINKABLE from EiffelBase, it seems that we can infer that two distinct linked list objects cannot share the same linkable element because all features with argument or result type, LINKABLE[T] are secret. This means that it is not possible to pass a reference to a linkable instance from a linked list instance to another linked list instances (other than itself) and thus no sharing can occur. This argument is not sound because of the following points:

- Newer versions of LINKED_LIST[T] might export new features with

signatures involving LINKABLE[T], e.g. a feature that returns a deep copy of the linkable element at the cursor position.

- References to linkable instances can be passed in as arguments and out as results of any feature with signature having an argument of type A or result of type A, where A is any ancestor or descendant of LINKABLE[T] e.g. ANY, CELL[T], BI_LINKABLE[T]. This occurs in several exported features of LINKED_LIST[T] since T can be any type (any descendant of ANY) and thus, for example, LINKED_LIST[LINKABLE[INTEGER]] is a valid derived type, with e.g. feature, item(i:INTEGER):LINKABLE[INTEGER]. It might be argued that the bodies of features of LINKED_LIST[T] cannot call the features of LINKABLE[G] on entities of type, T, but only those features applying to all types i.e. features defined in ANY, and furthermore this is confirmed statically by the type-checker. However, by using reverse assignment attempt, it is possible to bypass this restriction (for example, where T is an heir of LINKABLE[T]).
- By the selective export rules, descendants of LINKED_LIST[T] also have access to those features of LINKABLE[T] that are selectively exported. If descendants of LINKED_LIST[T] are defined that do allow sharing of linkable elements (e.g. LINKED_TREE[T]), some of the inherited features of LINKED_LIST[T] may fail to work together properly and may cause violations of their own assertions at run-time. Another problem is that since it is possible for two of the descendant instances to share linkable elements, they may behave in a way that is not compatible with two linked list instances (which cannot share linkables), destroying system correctness.

It might be argued that the above circumstances are pathological and would simply not occur in practice. However, while the original design of LINKED_LIST[T] is unlikely to contain such anomalies, later, modified versions and descendant classes could possibly introduce the above problems, especially if the maintainer or child class designer did not understand the encapsulation assumptions of LINKED_LIST[T].

2.3 Summary

The requirements of a language mechanism for defining components of an aggregation are given in the first column of Table 1, while the analysis (detailed in this section) of the suitability of existing mechanisms and techniques with respect to these requirements is summarized in the other 3 columns.

Component property	Inheritance	Expanded	Reference
Encapsulation	Yes	Yes	No
Information hiding	Yes	No	No
Arbitrary Creation	Yes	No	Yes
Polymorphic	No	No	Yes
Existence-dependent	Yes	Yes	No
Deferred type	No	No	Yes
Recursive type	No	No	Yes

Table 1 : Inheritance vs. Expanded attribute vs. Reference attribute.

Clearly, none of the techniques for implementing components examined in this

section fulfil all of the necessary requirements.

3. Object Ownership

We introduce the concept of object ownership, suggest syntax and semantics and discuss some possible generalizations and implementation considerations. We show that reference attributes with ownership restrictions (private attributes) satisfy all the properties derived from the requirements analysis of Section 2 and summarized in Table 1. We only show that encapsulation, information hiding and existence dependence are satisfied, since the other properties follow immediately from the last column of Table 1.

3.1 Syntax and semantics

The syntactic extension is the introduction of two new keywords, **`private`** and **`protected`**. These keywords are used as annotations (or declarations) describing the ownership of objects attached to entities[2] (attributes, local variables, formal arguments, *`Current`* and *`Result`*). An example of their use is given below.

The semantics is presented as a series of rules. Firstly, the rules governing object ownership (a run-time concept) are given. These are followed by the rules for entity proprietorship, including interpretation of static declarations of object ownership. An example is given showing how the concept of object ownership can be applied and demonstrating entity proprietorship declarations. The revised semantics for reattachment operations (creation, assignment and reverse assignment attempt) are then presented. In Section 3.1.1, we demonstrate that this semantics ensure that aggregates implemented using object ownership satisfy the properties of encapsulation and information hiding.

Object ownership is a run-time notion governed by the following rules:

- Every non-expanded (i.e. reference) object has an owner object (which could be Void).
- Expanded objects do not have owners.
- Ownership of an object is immutable, i.e. constant throughout its lifetime (a generalization that relaxes this condition is considered in Section 3.3).
- When the owner of an object dies, i.e. becomes garbage (unreachable from the root), the object itself dies also. Thus an object is existence-dependent upon its owner.

Entity proprietorship is the mechanism by which object ownership can be specified in software texts, and is defined by the following rules:

- Every entity of reference type has a proprietor object.
- Entities of expanded type do not have proprietors or proprietorship annotations.
- The proprietor of a public entity (the default) is Void.
- The proprietor of a private entity (keyword **`private`**) is the object in which it is declared.
- The proprietor of a protected entity (keyword **`protected`**) is the owner of the object in which it is declared.

The following example illustrates the new syntax, and shows how the ownership properties of linkable elements of linked lists (as described in Section 2.2) can be specified.

Example - Linked lists revisited

```
class LINKED_LIST[T]
feature
```

[2]The term *entity* is Eiffel terminology, and denotes a different concept from that used in the *entity-relationship (E-R)* model. In C++ terminology, an entity is (roughly) the lvalue of an expression (when it exists), and the object attached to an entity is (roughly) its rvalue.

```
     first : T
feature {NONE} -- secret
first_elem : private LINK[T]
put_linkable_left
(new : private LINK[T])
.....
end

class LINK[T]
feature
right : protected LINK[T]
put_right
(other : protected LINK[T])
....
end
```

Thus, a linked list is the owner of all its linkable elements, but not of the items in the list.

The relationship between object ownership and entity proprietorship is summarized by the following rule:

- The owner of an object attached to an entity must be the proprietor of that entity.

The proprietor object of an entity cannot be determined statically, but the entity to which the object is attached can be determined statically e.g.

```
class RECTANGLE
feature
top_left : private POINT
....
p : POINT, r : RECTANGLE
..... p:=r.top_left....
```

The proprietor of the entity, r.top_left, is the object attached to the entity, r.

Declarations of entity proprietorship and run-time object ownership obtain useful semantics, when the reattachment operations are redefined so as to restrict sharing of objects according to their ownership properties. The revised definitions for each of the reattachment operations is given below:

1. The creation call,
   `!..!writeable. ....`
   creates a new object of the appropriate class, sets its attributes to their defaults and then initializes them by calling the creation routine upon it. In addition, the owner of the newly created object is set to be the proprietor of the `writeable`.

2. The assignment operation,
   `writeable:= readable`
   behaves exactly as in Eiffel if the proprietor of the `writeable` is identical to the owner of the object attached to the `readable` (the proprietor of the `readable`), assuming that both entities are of reference type. If the proprietorship of the readable and writeable entities is different, then an exception is raised at run-time. If normal attachment were to take place, then the proprietors of the readable and writeable entities would be sharing the ownership of the object attached to both entities. Since the proprietorship of an entity cannot be determined statically, it is not possible to detect illegal attachments at compile-time. Instead, if an illegal attachment occurs at run-time, an exception is raised. This is analogous to a feature call on a void target.

3. The reverse assignment attempt is similarly redefined.

We might also introduce a private assignment attempt, which like the reverse assignment attempt, works exactly as assignment, when the ownership conditions are satisfied, but otherwise makes the writeable target entity become Void.

4. Whenever attachment of entities to objects occurs e.g. the attachment of formal argument entities to actual argument objects in a feature call, a run-time exception is raised if the proprietor of the entity is different from the owner of the object being attached to it.

3.1.1 Encapsulation and Information hiding

A private attribute is an attribute with a **`private`** ownership annotation. Each component (part) of a composite object (aggregate) is owned by the aggregate and

attached to at least one of its private attributes. So no object other than the aggregate is the proprietor of an entity that is attached to one of the components of the aggregate (by the restrictions on reattachment for objects and entities with ownership annotations). Hence no other object (except the aggregate) can access or modify a component of the aggregate except by calling a feature of the aggregate object. This shows that the requirements of information hiding and encapsulation are satisfied for this technique of modelling and implementing aggregations. Satisfaction of the other properties of Table 1 has already been shown.

3.2 Generalization - Ownership by a Private Club

The rules could be generalized as follows:

- An object can have multiple owners, and an entity multiple proprietors.
- An object can be shared amongst its multiple owners.
- An object dies when all its owners die.

Concrete syntax

```
e : share(e1,. .,en) T
```

meaning that the object attached to e is jointly owned by the objects currently attached to entities e1,. . ,en.

Attachment of an entity to an object is legal only if the set of owners of the object is identical to the set of proprietors of the entity.

3.3 Other generalizations

Ownership of an object can be mutable with changes of ownership being accomplished by an *acquire* command. For example:

```
writeable.acquire(readable)
```

means that the owner of the object (say *o*) attached to `readable` is reset to the proprietor of `writeable`, all existing references to o are reset to Void, and then:

```
writeable:=readable
```

is performed. It is not clear whether such a feature would be sufficiently useful in practice to justify the additional complexity. Other possible generalizations include allowing proprietorship annotations other than **`private`** and **`protected`** e.g. **`owner`**`(e)` meaning that the proprietor of the entity is the object attached to the entity, e.

3.4 Ownership monitoring as a compilation option

Run-time monitoring of object ownership and legality of assignments clearly incurs some overhead - an extra attribute (for the owner or collection of owners) for each object, and extra checking for each instruction involving reattachment of entities e.g. assignment, feature call with arguments etc. This overhead may be worthwhile during testing of the system, but unacceptable for the delivered system. This is analogous to run-time assertion monitoring and we propose the analogous solution of specifying ownership monitoring as a compilation option.

3.5 Language Support vs. Ownership Library

It might seem that we can model object ownership within the Eiffel language, instead of by language extension. The most elegant way to do so is by defining a class library with classes modelling ownership properties and restrictions. Although it is possible to define an attribute, *owner : ANY* to denote the owner of an object, it does not seem possible within Eiffel (unlike C++) to redefine the meaning of assignment and other reattachment operations (e.g. the attachment of formal argument entities to actuals in a feature call). This is because reattachment (like creation) is not a feature call with a target object but an operation with a target writeable entity and source object (see [Meyer 92] Chapter 20). Notice that creation itself cannot be redefined: the effect is always to create a new object and attach it to the writeable target entity. However, the initialization feature call on a newly created object, which is just a special kind of feature call (obeying creation rather than export

restrictions) can be redefined by redefining the initialization feature in a child class. Similarly, it is not possible to define the precondition assertion of reattachment as equality of the owner attributes of the source object and the proprietor 'attribute' of the target entity, because the latter is an entity and not an object and thus cannot have attributes.

4. Modeling aggregations

This section is not intended to demonstrate that all interesting properties of all aggregation structures can be captured using object ownership annotations, but simply to show that such annotations are a useful complement to other techniques and offer natural and expressive models for some examples.

This example (discussed comprehensively in [Civello 93b]) appears originally in one of the case studies (the Drawing Editor) of [Wirfs-Brock 93]. Drawings consist of collections of figures, some of which may be selected (some or all may be selectable). Figures cannot be shared between different drawings. Notice that FIGURE may well be a deferred class with effective children implementing different kinds of figures e.g. rectangles, ellipses, text boxes, tables. As explained in Section 2, it is not possible to implement deferred type components using inheritance or expanded type attributes.

```
class DRAWING
..
feature
shapes : private
      LIST[protected FIGURE]
selected : private
      LIST[protected FIGURE]
....
end
```

where

```
class LIST[T]
..
feature
      first : T
      ith (i : INTEGER) : T
      ...
end
```

Notice that **protected** FIGURE is not a type, but a type (FIGURE) together with an ownership annotation (**protected**). Type derivation is the usual syntactic replacement of formal generic parameter by actual type (FIGURE), except that the ownership annotation is also substituted. The effect is to define a list of figures, the items of which are owned by the drawing object, as required. Similarly, both the list of selected figures and the items in that list are all owned by the drawing object.

5. Related Work

None of the major commercial OO programming languages (Eiffel, C++, Smalltalk, Objective-C and Ada-93) provides language support for object ownership and we are not aware of any OOPL providing such support.

Many OO development methods have a concept of and notation for aggregation structures (sometimes called part-of hierarchies) in OO models and designs. As well-established methods have matured and second generation methods (e.g. BON [Waldén 94] and Syntropy [Cook 94]) developed, a better understanding of the different roles and properties ascribed to aggregation structures in different contexts and examples has emerged (see e.g. [Waldén 94]p.70-71 and [Cook 94]p. 38-40). The fact that Eiffel expanded types do not provide a general solution to the problem of modelling aggregations is noted in [Waldén 94]p.80. [Civello 93a] is a deep analysis and comprehensive classification of the roles and properties of aggregation structures in object-oriented modeling, analysis and design. Object ownership enriches modeling by capturing some of the properties of aggregations (e.g. life-time dependency) precisely. [Dong 95] gives an analysis of the related concept of object containment.

The notion of object ownership simplifies reasoning about linked structures by restricting aliasing and sharing of component objects, thereby preventing the possibility of interference. The presence of aliasing is regarded by the formal methods community as one of the major challenges of specifying and verifying object systems[Duke 94, Hogg 91a]. We briefly relate our work to other techniques developed for restricting aliasing to simplify reasoning.

[Hogg 91b] introduces the concept of islands and bridges to restrict aliasing in object-oriented programs and thereby simplify reasoning. Hogg introduces three access modes for entities: *read*, *unique* and *free*. *Read*-only entities (like *const* entities in C++) provide read-only (side-effect-free) access to the objects they denote. *Unique* entities are always attached to Void or unshared objects (only one static reference in the entire system). *Free* entities can only denote objects with no static reference in the entire system. For example, creation expressions (i.e. queries returning newly created objects) are free entities. Hogg specifies restrictions on the use of entities for each of these modes to ensure that the entities satisfy the properties of their declared modes (like the rules for 'const correctness' in C++). In contrast to our rules for object ownership, conformance to these restrictions can be checked statically.

An island is the transitive closure of a set of objects accessible from a bridge object. A bridge is the sole access point to a set of instances that make up an island. The bridge object of an island must be an instance of a bridge class. A bridge class must satisfy the following restriction:

> every formal argument of each exported feature and the result of all exported queries must have (at least) one of the three access modes: read, unique or free.

Hogg suggests implementing collections as islands with container objects as bridges. However, the class LINKED_LIST is not a bridge class, because (for example) the result of the query, *ith*, has none of the access modes: read, unique or free. In addition the *right* attributes of the linkable elements and the arguments to their *put_right* command are neither unique nor free in general. For example, the cursor attributes, *active*, *previous* and *next* are all aliased to *right* attributes of linkables. Internal sharing also occurs in other linked structures such as circular lists and doubly linked lists.

Formal inference systems for reasoning about aliasing in object systems are given in [America 90, Wills 93], but these are very complex.

[Jones 92] presents inference rules for reasoning about concurrent object-based systems. Some of these depend upon 'private references' to prohibit interference and simplify reasoning. One of his examples involves a priority queue implemented as a linked list. However, Jones does not give detailed formal semantics for private references.

6. Conclusions

We have proposed a concept of object ownership and outlined a syntax and semantics for extending Eiffel to support this concept directly.

We have shown two situations for which limiting the ownership of objects is useful:

1. specifying, implementing and verifying linked structure data types.
2. natural, expressive modeling of complex aggregation structures.

In addition, we have shown how the introduction of object ownership helps prevent overloading of the inheritance mechanism (for code reuse in addition to polymorphism) and expanded types (for components as well as value types),

thereby making the language mechanisms more orthogonal.

However, our analysis falls short of the complete account of object ownership necessary to judge whether or not it is worthwhile to extend the Eiffel language standard to support this concept (see [Meyer 92] Appendix B.10, p. 508). In particular, we have not considered the possible interaction of object ownership with all other Eiffel constructs and mechanisms (e.g. the `Result` of a **once** function cannot be private, can protected entities be redefined as private in heirs?). Also, we do not claim to have made the optimal choices for reattachment in the presence of ownership annotations. For example, it may be advantageous to apply the semantics of reattachment given in Section 3.1 only when the writeable entity is an attribute or *Result*, and use conventional reattachment (i.e. don't check proprietorship of the target) when the target is a local entity so that objects can have temporary references to objects they don't own for the duration of a feature call. Also, we have made no judgement about which of the generalizations suggested in Section 3 might be worthwhile. It is clear that much work is still required (e.g. developing a prototype implementation of a pre-processor for the extended language, resolving the problems raised above) on the topic of object ownership and its support in Eiffel and other OOPLs.

References

[America 90] P. America and F. de Boer. A sound and complete proof system for SPOOL. Technical Report 505, Philips Research Labs, May 1990.

[Civello 93a] F. Civello. Roles for composite objects in object-oriented analysis and design. OOPSLA'93, ACM Press, pages 376-393, 1993.

[Civello 93b] F. Civello. Roles for composite objects in object-oriented analysis and design. PhD. thesis, University of Brighton, 1993.

[Cook 94] S. Cook and J. Daniels. Designing Object Systems, Prentice Hall 1994.

[Dong 95] J.S. Dong and R. Duke. The geometry of Object Containment. Object Oriented Systems, vol. 2, no. 1, 1995.

[Duke 94] R. Duke. Do Formal Object-Oriented Methods have a future? Keynote TOOLS Pacific, Melbourne 1994, in TOOLS 15 (Mingins and Meyer, eds.), Prentice Hall, pages 273-280, 1994.

[Hogg 91a] J. Hogg, D. Lea, A. Wills, D. de Champeaux and R. Holt. The Geneva convention on the treatment of object aliasing, in Follow-up Report on ECOOP '91 Workshop W3: Object-oriented Formal Methods, pages 11-16, 1991.

[Hogg 91b] J. Hogg. Islands : Aliasing Protection in Object-Oriented Languages, OOPSLA'91, ACM Press, pages 271-285, 1991.

[Jones 92] C.B. Jones. An Object-Based Design Method for Concurrent Programs. Technical Report UMCS-92-12-1, University of Manchester, 1992.

[Kent 95] S. Kent and I. Maung. Quantified Assertions in Eiffel, Proceedings of TOOLS Pacific 95, Prentice Hall, December 1995.

[Meyer 88] B. Meyer. Object-oriented Software Construction. Prentice Hall 1988.

[Meyer 92] B. Meyer. Eiffel, the Language. Prentice Hall 1992.

[Meyer 94] B. Meyer. Reusable Software - the base OO component libraries. Prentice Hall 1994.

[Waldén 94] K. Waldén and J.-M. Nerson. Seamless Object-Oriented Software Architecture, Prentice Hall 1994.

[Wills 93] A. Wills. Reasoning about aliasing. unpublished manuscript, 1993.

[Wirfs-Brock 90] R. Wirfs-Brock, B. Wilkerson and L. Wiener, Designing Object-Oriented Software, Prentice Hall 1990.