J.S. Dong and R. Duke. The Geometry of Object Containment. Object-Oriented Systems (OOS) journal 2(1):41-63, Chapman Hall, 1995. (ISSN: 09699767)

The Geometry of Object Containment[1]

Jin Song Dong **Roger Duke**

Software Verification Research Centre
Department of Computer Science
University of Queensland, Australia 4072

jason@cs.uq.oz.au rduke@cs.uq.oz.au

Keywords: formal specification, object containment, recursive structures

Abstract

In object-oriented systems, it is often the case that an object will have an attribute whose value identifies (points or refers to) some other object in the system so that the identified object can be sent messages. The association between objects determined by the object references in a system will generally result in a complex structure whose design and specification is a crucial part of the development and implementation of the system. The aim of this paper is to look at ways of capturing formally object reference structures that occur frequently in object-oriented systems. For example, consider a system consisting of car and wheel objects where each car has attributes referencing its wheel objects. If wheels are not shared between cars, distinct car objects will reference distinct wheel objects. In this paper we distinguish such object references by saying that a car object (directly) *contains* the wheel objects it references. The nature of the contained object references as suggested by this example is captured within a formal framework and incorporated into the Object-Z specification language. Object containment is then compared with related ideas found in object-oriented programming languages. The distinction between object containment and the notion of object ownership (control) is also discussed. Finally, a generalised notion of object containment is investigated.

1 Introduction

In object-oriented systems, references between objects are maintained so as to facilitate inter-object communication[Boo91, GR89, Mey88]. For example, consider a banking system consisting of account objects, customer objects and bank objects. In such a system, a customer object will have attributes whose values reference account objects. These references enables customers to operate their accounts (e.g. to make deposits, withdraws, etc.). If a bank permits shared accounts, several customers may even reference the same account. In addition, an account object may well have

[1] In *Object-Oriented System (OOS)* 2:41-63, Chapman & Hall 1995

an attribute whose value is a reference to a customer object, so that access to the account can be authorised. Furthermore, a bank object will have attributes whose values reference account objects, so that the bank can operate the accounts for the purpose of adding charges or changing credit limits.

The association between objects determined by the object references in a system will generally result in a complex structure whose design and specification is a crucial part of the development and implementation of the system. The aim of this paper is to look at ways of capturing formally object reference structures that occur frequently in object-oriented systems.

As an example, suppose that in a banking system no account is shared between banks. In this case it would follow that any account referenced by one bank is distinct from any account referenced by a different bank. When giving a formal specification of this banking system, such an important structural property of the object references should be clearly captured by the specification, ideally as a global system invariant.

As another example, consider a system consisting of car objects and wheel objects. Each car will have attributes that reference its wheel objects. If wheels are not shared between cars, distinct car objects will reference distinct wheel objects.

In this paper we formally investigate the general nature of the object references implicit in the last two examples. The properties of such object references are captured within a formal framework and incorporated into the object-Z[DKRS91, Ros92] specification language as predicate rules (resembling in flavour the axioms of combinatorial geometric structures such a s projective planes).

The example above of the car and its wheels suggests a notion of geographical location (a car physically contains its wheels) and for this reason we refer to the resulting object-references as *containment*, i.e. a car object is said to (directly) *contain* the wheel objects it references. As the banking system illustrates, however, the structure of object containment also arises in situations where the objects have no relevant geographical location: it would be inappropriate to think of an account object as being physically contained within a bank object, but nevertheless, because distinct banks reference distinct accounts, a bank object is said to (directly) contain the account objects it references. In the paper this notion of object containment is precisely characterised by its (geometric) properties.

In general, some attributes of an object will reference contained objects, and other attributes will not. For example, in addition to the references to its contained wheel objects, a car object may well reference a person object corresponding to the owner of the car. We would not expect the person reference to denote object containment as a person may well own several cars. That is, in general an object 'has a' set of references to other objects, only some of which may be 'contained' references in the sense defined in this paper.

Various notions of object association, such as aggregation and composition, have been discussed in the literature[CY91, Civ93, DT93, HS92, Lif93, Nie93, Ode92]. Our notion of containment has features in common with them, but is not identical to any. A notion of containment is also defined by Kilov and Ross[KR94] in extended Object-Z and used (as a hierarchical subordination) in a way different from the treatment in this paper.

In this paper, the properties implied by object containment are modelled by a constraint relationship between objects. Two examples are presented in Section 2 to demonstrate that this containment relationship can be captured explicitly in Object-Z by predicates in the state invariant of a class. However, when a system is large and complex, capturing the properties of the containment relationship explicitly in this way is cumbersome. Therefore, in Section 3 an extension of the Object-Z notation is introduced to capture directly the geometric notion of object containment.

The notion of object containment is also partially supported by some object-oriented programming languages. Object containment in object-oriented programming languages such as Eiffel[Mey92] (*expanded* class type) and C++[Str86] (object type) is discussed and compared to our notion in Section 4.

In many object-oriented systems, object containment is closely related to object ownership: an object will have exclusive control of any object it contains. However, in this paper we view object containment as a purely geometric notion, quite distinct from the issue of object ownership. An example in Section 5 illustrates this.

Objects may overlap or share contained objects, e.g. two rooms may share a wall in a building. An object may also contain only part of another object, e.g. a street may pass though and hence be partially contained by several suburbs. In Section 6 this generalised view of object containment is illustrated and formally captured in Object-Z.

Finally, the geometry of object containment can be applied to simplify the specification of abstract recursive structures, such as trees and directed acyclic graphs. Examples are presented in Section 7.

For those readers not familiar with Object-Z, a glossary of the notation used in this paper, together with a brief introduction to Object-Z is given in an appendix. Further details are given in [DKRS91] and [Ros92].

2 Capturing the Properties of Containment

In this section the properties of object containment are stated precisely. The notion of object containment informally introduced in the last section suggests a forest-like geometric relationship between contained objects. As a motivating example, in Figure 1 u, v, w, x, y and z denote objects where u directly contains w and x, and similarly w directly contains y and z, but u and v are not related by containment, and neither are w and x. In this case object u *indirectly* contains y and z.

Figure 1 is based on an idea of geographical object location. However, object containment can arise in systems where the objects are not related geographically. Nevertheless, the figure suggests two geometric ideas that characterise the general notion of object containment[2]:

(1) an object cannot directly or indirectly contain itself; and

(2) an object cannot be directly contained within two distinct objects.

[2]In this paper, the term 'geometry' is used in a combinatorial sense. A geometric structure is defined by the rules that determine whether or not one object 'contains' another. Like projective geometries, no geographical notion of physical location is implied by the term.

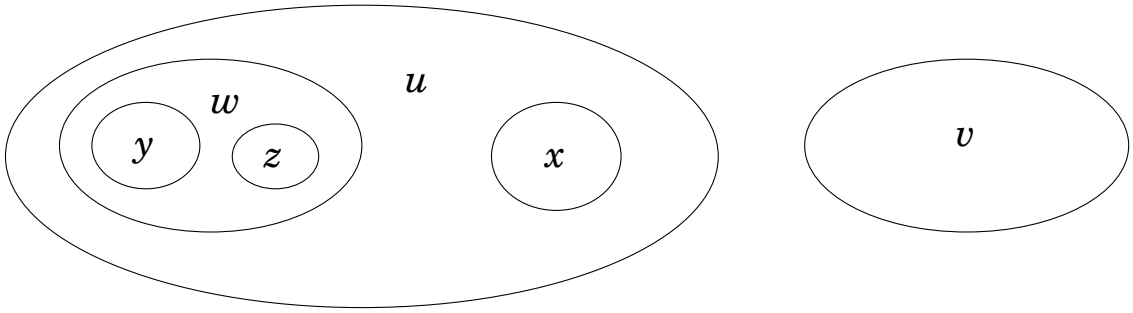


Figure 1: The geometry of object containment

To capture these ideas formally, let $\mathbb{O}$ denote the universe of all object identities in a system[DD93], and let

$$dcon : \mathbb{O} \leftrightarrow \mathbb{O}$$

denote the relation whereby

$$ob_1 \ \underline{dcon} \ ob_2$$

if and only if object ob_1 has a reference to a $\underline{d}$irectly $\underline{con}$tained object ob_2. Put another way, $dcon$ is the set of all those ordered pairs (ob_1, ob_2) of (identities of) objects in the system where ob_1 directly contains ob_2.

The first condition above requires that

$$\nexists ob : \mathbb{O} \bullet ob \ \underline{dcon^+} \ ob$$

where $dcon^+$ is the transitive closure of $dcon$. The second condition requires that

$$dcon^\sim \in \mathbb{O} \nrightarrow \mathbb{O}$$

(i.e. the inverse of the relation $dcon$) is a partial function.

In any system, the relation $dcon$ is not static; it will change dynamically if object relocation is permitted, i.e. if there are operations in the system that affect the containment geometry. This is discussed further in Section 3.4.

When specifying an object-oriented system using Object-Z, the above two properties of object containment can be captured explicitly by class invariants, as is illustrated in the following two examples.

2.1 Example: Terminal Location

Consider the situation where a campus consists of a set of buildings, with each building containing a set of rooms and each room containing a (possibly empty) set of terminals. A specification in Object-Z would be

Terminal

[details of state and operations omitted]

Room

$ts : \mathbb{P}\ Terminal$

[details of operations omitted]

Building

$rs : \mathbb{P}\, Room$

$\forall\, r_1, r_2 : rs \bullet$
$\quad r_1 \neq r_2 \Rightarrow$
$\quad\quad r_1.ts \cap r_2.ts = \varnothing$

[details of operations omitted]

Campus

$bs : \mathbb{P}\, Building$

$\forall\, b_1, b_2 : bs \bullet$
$\quad b_1 \neq b_2 \Rightarrow$
$\quad\quad b_1.rs \cap b_2.rs = \varnothing$
$\quad\quad \forall\, r_1 : b_1.rs;\ r_2 : b_2.rs \bullet$
$\quad\quad\quad r_1.ts \cap r_2.ts = \varnothing$

[details of operations omitted]

The class invariant of the *Building* class captures the idea that no terminal can be in two distinct rooms in a building. Similarly, the class invariant of the *Campus* class captures the idea that no room can be in two distinct buildings of the campus. Furthermore, despite the fact that the predicate of the *Building* class states that no terminal can be in two distinct rooms, as this applies only to the rooms of a given building it says nothing about rooms in distinct buildings. Hence the predicate

$$\forall\, r_1 : b_1.rs;\ r_2 : b_2.rs \bullet r_1.ts \cap r_2.ts = \varnothing$$

needs to be conjoined to the predicate of the *Campus* class.

Clearly, capturing the properties of object containment explicitly in this way is cumbersome, particular if the system is large and complex. We would like to be able to give a global invariant that captures directly the condition that distinct rooms anywhere contain distinct terminals, and distinct buildings anywhere contain distinct rooms. The condition that distinct rooms contain distinct terminals, for example, is not an internal invariant of the *Room* class, but rather an invariant of any system containing room objects; nevertheless, it would be convenient to be able to attach such global conditions directly to the *Room* class. A way of doing this is given in Section 3.

2.2 Example: Russian Dolls

Object containment is sometimes an important property of recursive structures. For example, consider the situation of Russian dolls. Each doll is either solid or hollow, and each hollow doll contains another doll that is itself solid or hollow. The fundamental property of this set of recursively embedded dolls is that no doll directly or indirectly contains itself.

An object-oriented specification in Object-Z would be

$$Doll \mathrel{\widehat=} Solid \cup Hollow$$

Solid

$dolls : \mathbb{P}\, Doll$

$dolls = \varnothing$

Hollow

$inside : Doll$
$dolls : \mathbb{P}\, Doll$

$dolls = \{inside\} \cup inside.dolls$
$self \notin dolls$

The attribute *inside* identifies the doll directly contained within a hollow doll; the attribute *dolls* denotes the set of all dolls directly or indirectly contained within a doll. By specifying *Doll* to be the union[3] of the classes *Solid* and *Hollow*, the doll identified by *inside* is either solid or hollow.

The predicate $self \notin dolls$ of the class *Hollow* ensures that a doll does not directly or indirectly contain itself. Note that *self* represents an object's own identity (see [DR93] for details).

This specification takes an object-oriented view, modelling each doll as an object with a unique identity. It could be argued that it is more complex than a functional recursive specification using the Z free type[Smi91]. However, when object containment is captured by global predicates in Section 3, the object-oriented specification mimics in style the definition using free types.

3 Capturing the Geometry of Containment

In the examples in the last section, the properties of object containment were formally captured in Object-Z by writing explicit class invariants. There are several consequences of that approach:

- The invariants that result can be complex, particularly as object containment is often a significant aspect of a system's design.
- An appropriate invariant needs to be placed in each relevant class. As the invariant is capturing the same concept of object containment in each case, the specification can become repetitious.
- Only the *properties* that follow from object containment are being captured by the invariant. The geometric notion as to which objects are actually contained within a given object is not explicitly stated.

In this section, specific notation is introduced into Object-Z to capture directly the geometric relationship of object containment. This notation enables the specifier to state explicitly, as part of the class specification, which objects will be directly contained within an object of that class.

To be precise, let each class have an implicitly declared attribute

$$dcontain : \mathbb{P}\,\mathbb{O}$$

where the value of *dcontain* is the set of directly contained objects. Then in any system the relation

$$dcon : \mathbb{O} \leftrightarrow \mathbb{O}$$

introduced in Section 2 is determined by

$$\forall\, ob_1, ob_2 : \mathbb{O} \bullet \\ ob_1 \mathrel{\underline{dcon}} ob_2 \Leftrightarrow ob_2 \in ob_1.dcontain.$$

[3]The notion of class union is discussed in detail in [DD93].

The properties of the relation *dcon* (as stated in Section 2) imply invariant conditions on the system that need not be stated explicitly. In terms of the attribute *dcontain* these conditions are:

$$\begin{array}{l} \nexists s : \mathrm{seq}\,\mathbb{O} \bullet \\ \quad \#s > 1 \\ \quad \forall i : 1 \,..\, \#s - 1 \bullet s(i+1) \in s(i).dcontain \\ \quad s(1) = s(\#s) \end{array} \qquad [\mathsf{dc1}]$$

(i.e. no object directly or indirectly contains itself)

$$\begin{array}{l} \forall ob_1, ob_2 : \mathbb{O} \bullet \\ \quad ob_1 \neq ob_2 \Rightarrow ob_1.dcontain \cap ob_2.dcontain = \varnothing \end{array} \qquad [\mathsf{dc2}]$$

(i.e. no object is directly contained in two distinct objects). The two predicates **dc1** and **dc2** are global invariants of any Object-Z specification.

3.1 Examples Revisited

With this notation the specification of both the terminal-location and Russian-dolls systems is significantly simplified.

Terminal Location

Terminal

[details of state and operations omitted]

Room

$ts : \mathbb{P}\, Terminal$

$dcontain = ts$

[details of operations omitted]

Building

$rs : \mathbb{P}\, Room$

$dcontain = rs$

[details of operations omitted]

Campus

$bs : \mathbb{P}\, Building$

$dcontain = bs$

[details of operations omitted]

Russian Dolls

$Doll \mathrel{\widehat=} Solid \cup Hollow$

Solid

$dcontain = \varnothing$

Hollow

$inside : Doll$

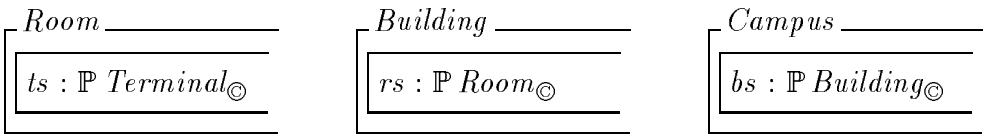
$dcontain = \{inside\}$

In both these examples, there are implicit class invariants that follow directly from the properties of *dcontain* stated above. Indeed, from these properties the explicit class invariants given in Sections 2.1 and 2.2 can be deduced.

The convention is adopted that no mention of *dcontain* need be made if there are no contained objects. Therefore the predicate $dcontain = \varnothing$ can be omitted from the class *Solid*.

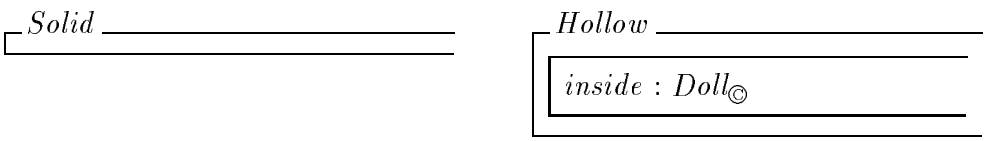
3.2 A Notational Simplification

If the role of an attribute is to always identify directly contained objects, this can be indicated when the attribute is declared by appending a subscript '$_{\copyright}$' to the appropriate type. This removes the necessity to write an explicit predicate involving *dcontain*. For example, adopting this syntactic convention, the relevant classes in the specification of the terminal location system become



while the specification of the Russian-dolls system become

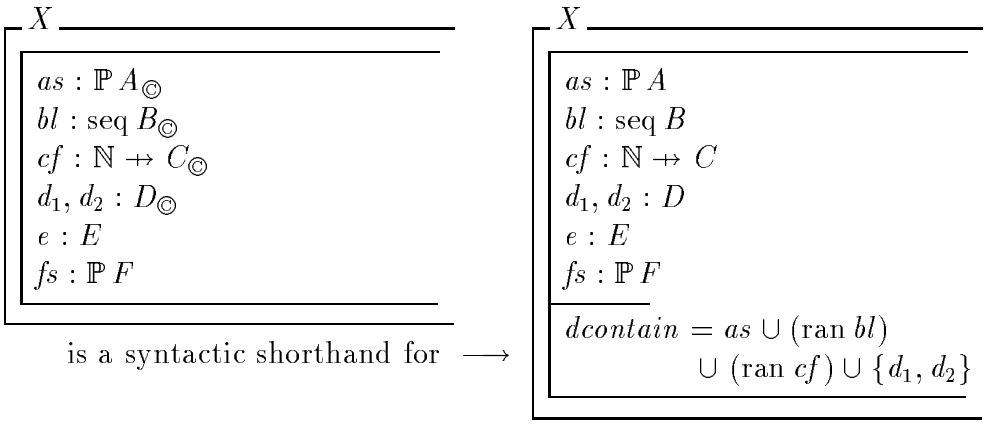
$$Doll \mathrel{\widehat=} Solid \cup Hollow$$



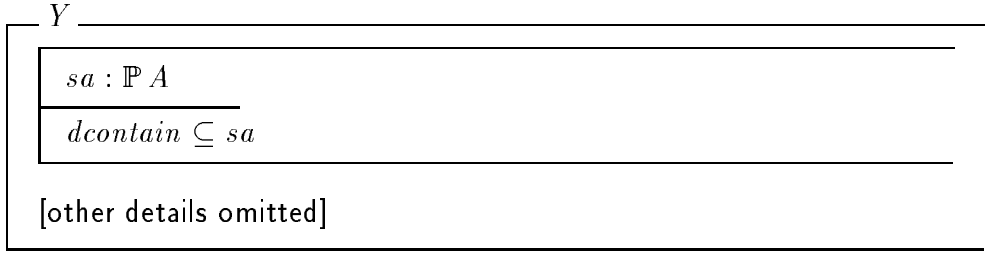
The subscript '$_{\copyright}$' is appended to the type of the attribute rather then the attribute itself because the attribute may identify a complex data structure rather than an object reference. For example, the declaration

$$ts : \mathbb{P}\, Terminal_{\copyright}$$

in the *Room* class declares *ts* to be a set, not an object reference. From its type, *ts* is a set of references to objects of class *Terminal*; the $_{\copyright}$ implies these references are to contained objects. Type declarations involving $_{\copyright}$ can be converted into declarations that directly use the attribute *dcontain* instead; for instance:



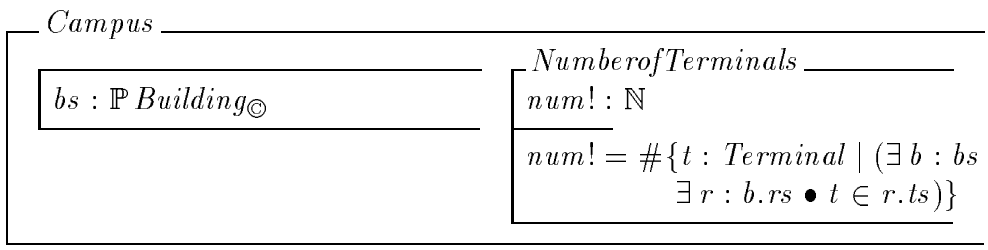
Notice that a syntactic simplification using $\copyright$ cannot always be applied, e.g. suppose A is some class and Y is a class defined by



The declaration $sa : \mathbb{P}\, A_{\copyright}$ would not be a correct simplification in this case. However this situation, when the value of *dcontain* is not explicitly determined, does not often arise when modelling real systems.

3.3 Indirectly Contained Objects

Although the introduction of the attribute *dcontain* is sufficient to capture the above properties of containment, it is often useful to explicitly identify all those objects that a given object directly or indirectly contains. For example, considering the terminal-location system, suppose an operation exists to output the number of terminals which are contained in the campus. The *Campus* class would then be



The predicate of the operation *NumberofTerminals* can be captured more easily if each class has an implicitly declared attribute

$$contain : \mathbb{P}\, \mathbb{O},$$

where the value of *contain* is the set of directly or indirectly contained objects, i.e. in terms of the relation *dcon* introduced in Section 2,

$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1 \in o_2.contain \Leftrightarrow o_2\ \underline{dcon^+}\ o_1.$$

To be precise, every class has an implicit class invariant

$$\begin{aligned} contain = \{ob : \mathbb{O} \mid \exists\, s : \mathrm{seq}_1\, \mathbb{O} \bullet\ & \\ & s(1) \in dcontain \\ & s(\#s) = ob \\ & \forall\, i : 1 \,..\, \#s - 1 \bullet s(i+1) \in s(i).dcontain\}. \end{aligned}$$

The predicate of the operation *NumberofTerminals* can now be written as

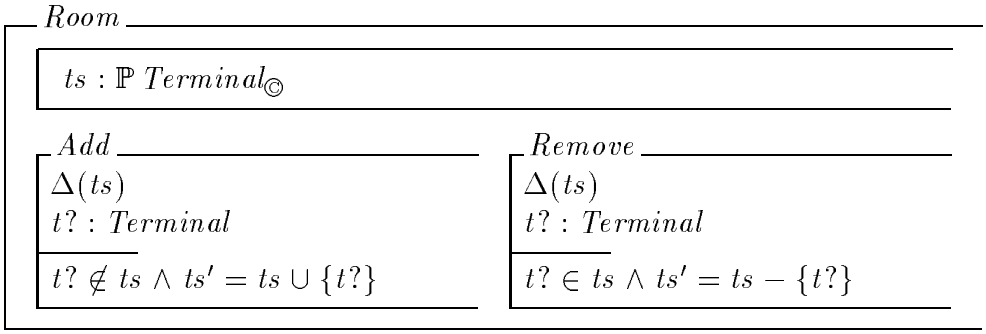
$$num! = \#(contain \cap Terminal).$$

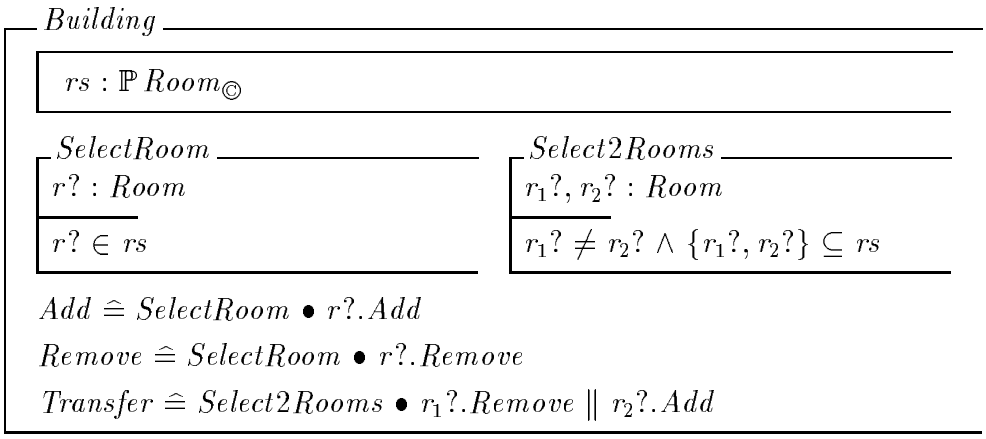
Notice that in terms of the attribute *contain*, the global invariant **dc1** becomes

$$\forall\, ob : \mathbb{O} \bullet ob \notin ob.contain.$$

3.4 Object Relocation

The geometry of object containment of a system is dynamic because a contained object can relocate from one container to another in the system. For example, considering again the terminal-location system, suppose operations exist to add a terminal to a room, to remove a terminal from a room and to transfer a terminal from one room to another inside a building. The appropriate part of the system specification would then become





It is to be understood that the Δ-list of the operations *Add* and *Remove* in the *Room* class implicitly includes the attributes *dcontain* and *contain* as these values are subject to change whenever the value of the state variable *ts* changes[4].

Notice that the implicit conditions implied by the geometry of object containment will be maintained at all times[5]. For instance, it will be the case, even although it has not been stated explicitly within the *Add* operation schema that the new terminal added to the room is not already in any other room on the campus.

In some cases, a contained object may be a fixed component of its containing object, i.e. object relocation may not be possible. For example, a room is usually a fixed component of a building. This is implicitly captured by having no operation that can change the attribute *rs* in the class *Building*.

[4]In effect, the attributes *dcontain* and *contain* are *dependent* attributes implicitly occurring in every Δ-list (for details see [DR93]).

[5]In Z and Object-Z the state invariant must hold before and after each operation.

4 Containment in Programming Languages

Some object-oriented programming languages support a view of object containment. In Eiffel, for instance, if the type of an attribute is an *expanded* class, the value of the attribute will be an actual object rather than an object reference. In effect, an attribute of *expanded* class type denotes a contained object. A consequence of this is that although the internal values of a contained object can be updated directly, relocation is not possible: the contained object is treated as a fixed component. Furthermore, both value semantics and reference semantics for objects needs to be defined. This contrasts with the approach adopted in this paper where reference semantics for objects is uniformly maintained. Not only does this simplify the Object-Z semantics, but it permits the relocation of contained objects.

To illustrate this distinction, consider a car that contains four wheel objects and references an owner object. In Eiffel this would be [6]

class Car **feature**
$w1, w2, w3, w4$:**expanded class** Wheel
owner: Person
...
end - - class Car

while in Object-Z it would be

Car

$w1, w2, w3, w4 : Wheel_{©}$
$owner : Person$

$\#\{w1, w2, w3, w4\} = 4$

...

The *Car* class invariant states that the four wheels are distinct.

Pictorially, the Eiffel and Object-Z models are given in Figure 2, where a dotted or solid arrow denotes an object reference, with the solid arrow denoting a reference corresponding to containment. In Object-Z, operations can be specified to allow a wheel of a car to be switched to a different position or even replaced. However, in Eiffel the use of the *expanded* class type means that such operations are not possible.

The value semantics of the Eiffel *expanded* class type also ensures that no aliasing[7] is possible for any object of such a class. If the source (right hand side) of an assignment is an object of *expanded* class type then the value of the source object is copied to the target; a reference to the source object is not copied. A consequence is that, in Eiffel, an object is the unique client of any object of expanded class type it contains; in effect, a client has exclusive control of its contained objects. This provides for aliasing protection, although with this approach the notions of control and containment are not distinguished; this issue is discussed further in Section 5.

[6] A similar example can be found in [Mey92].

[7] Aliasing occurs when an object can be accessed in more than one way, see [dCLF93, Hog91, Mey88].

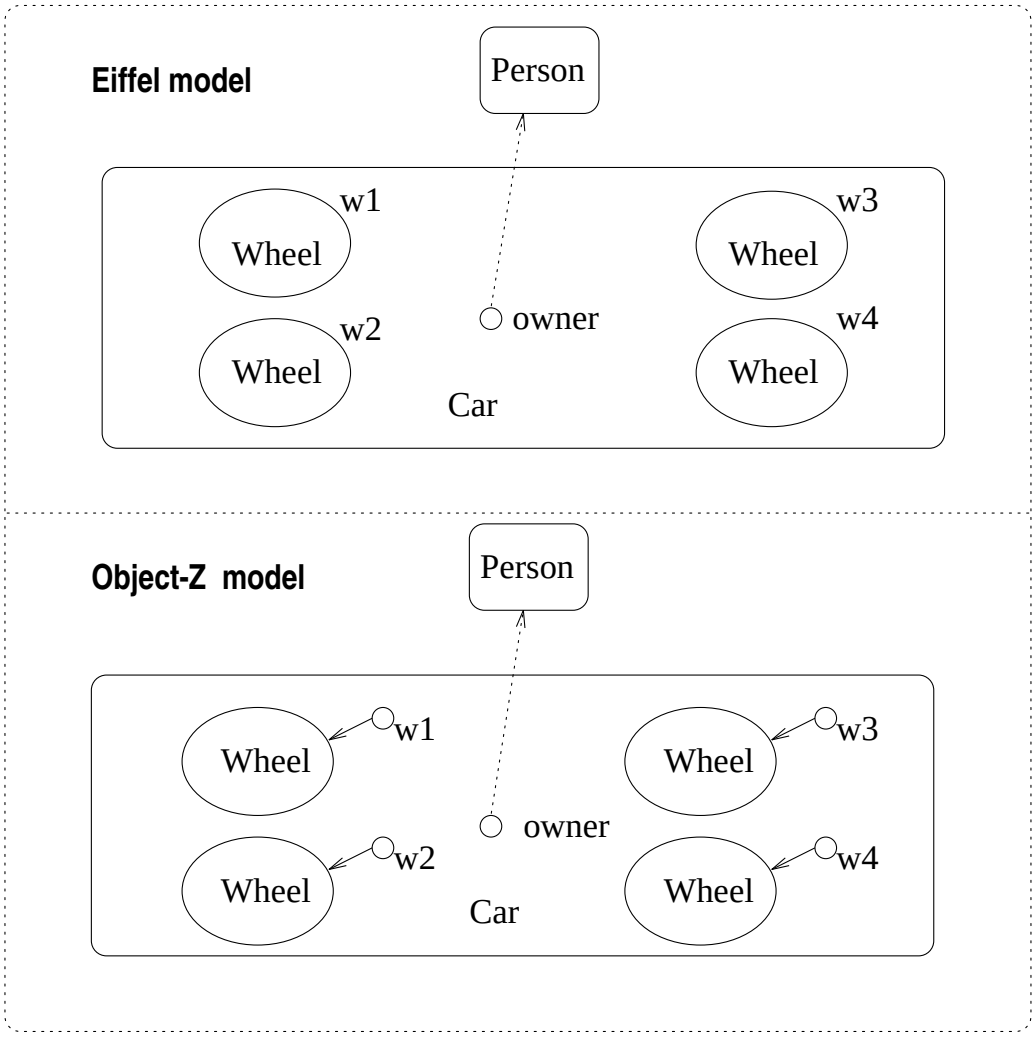


Figure 2: Containment in Eiffel and Object-Z

C++ has a notion of object containment similar to that of Eiffel except it makes object containment the default class type and containment does not imply unique control; i.e. C++ does not allow the relocation of a contained object from one container to another, but it does allow a contained object to be referenced by objects other than its containing object. In C++, the car example would be

```
class Car {
Wheel w1,w2,w3,w4
Person *owner
...
}
```

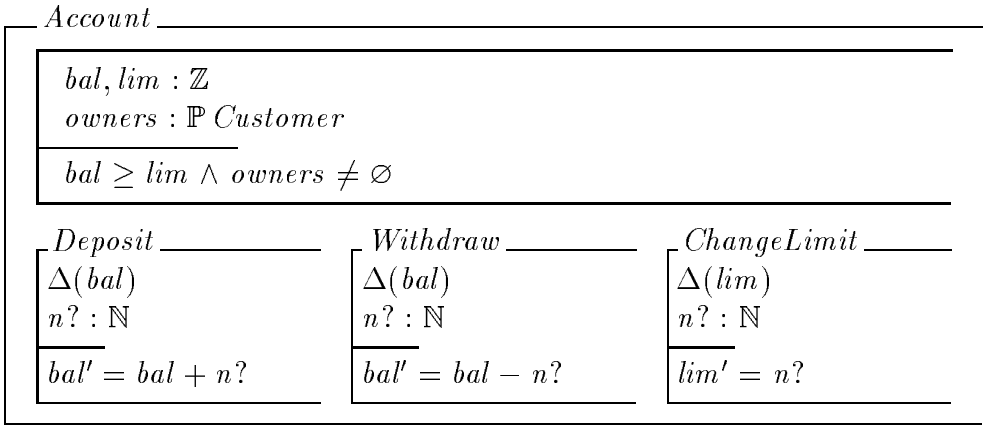
5 Containment and Control

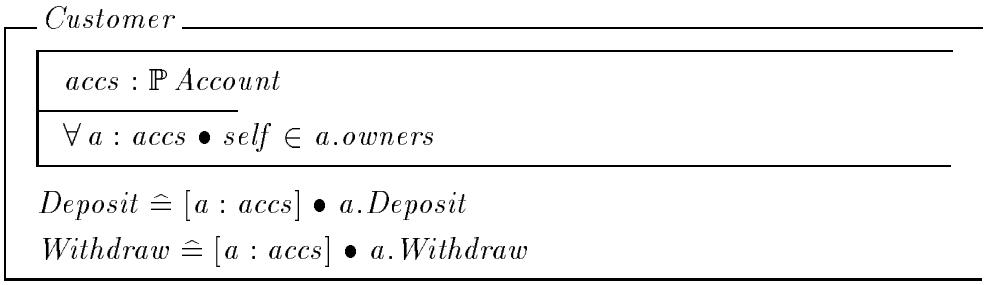
In object-oriented systems it is sometimes the case that a contained object is actually owned by the containing object, i.e. the containing object has exclusive control of

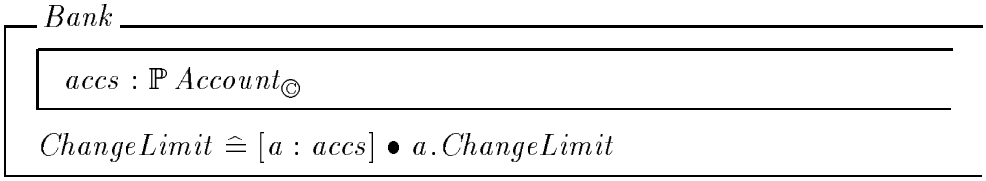
the contained object, e.g. as in Eiffel. However, the notions of containment and control are in general quite distinct and should not be confused in system modelling: containment is concerned with the relative geometry of objects; control is concerned with access, i.e. the right of one object to send a message to another object. Often within object-oriented systems the case arises when one object contained within another can be sent messages by a third object elsewhere in the system. This is illustrated in the following example.

5.1 Example: A Banking System

A banking system consists of account objects, customer objects and bank objects. An account has a balance, a limit below which the balance must not fall and a set of owners who can operate the account by making deposits or withdraws. Each account is contained within a bank (i.e. an account is referenced by a unique bank) which is able to change the limit of the account. We shall suppose that an account may be shared between customers. In Object-Z this becomes







A banking system consists of a set of banks and a set of customers.

BankingSystem

$banks : \mathbb{P}\, Bank$
$customers : \mathbb{P}\, Customer$

$\bigcup\{b : banks \bullet b.accs\} = \bigcup\{c : customers \bullet c.accs\}$

$Deposit \mathrel{\widehat=} [c : customers] \bullet c.Deposit$
$Withdraw \mathrel{\widehat=} [c : customers] \bullet c.Withdraw$
$ChangeLimit \mathrel{\widehat=} [b : banks] \bullet b.ChangeLimit$

The state invariant of this class ensures that the accounts contained in *banks* are precisely those accounts owned by *customers*.

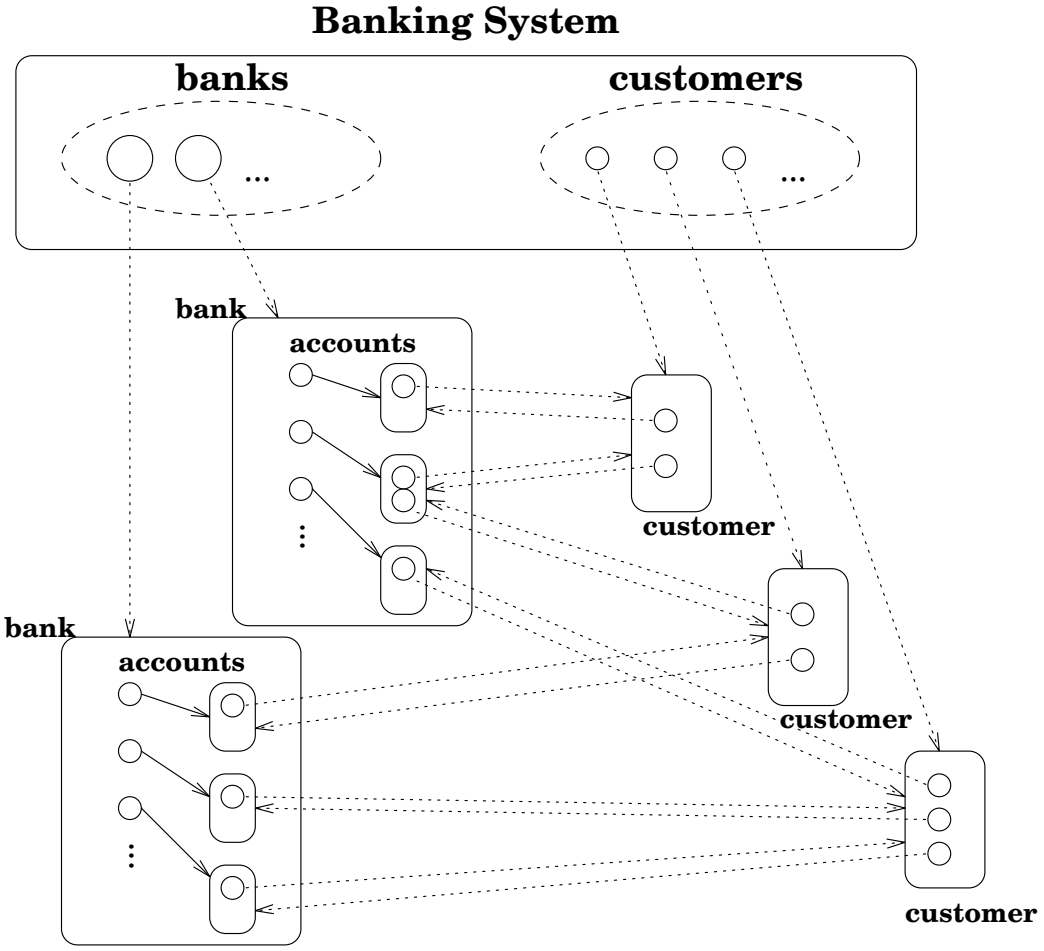


Figure 3: A banking system

Because accounts are contained in banks, the set of accounts is partitioned between the banks, i.e. each account is uniquely associated with a bank. However, customers control the balance of accounts owned by them, while the banks control only the limit of the accounts they contain. The object-reference structure of this banking system is illustrated in Figure 3 (where a dotted or solid arrow denotes an object

reference, with the solid arrow denoting a reference corresponding to containment).

6 Modelling Shared Containment

The geometric notion of object containment considered so far in this paper has been that of *unique* containment, i.e. an object cannot be directly contained within two distinct objects. However, this is not the only containment geometry found within real systems. In general, objects may overlap and share contained objects, e.g. two rooms may share a wall in a building. Furthermore, in some systems an object may contain only part of another object, e.g. a street may be located in several suburbs and hence is partially contained by each of the suburbs that share it. That is, property (2) in the early part of Section 2 holds for unique containment, but not for a more general notion of shared containment.

The geometric complexities that arise with this notion of shared containment are illustrated in Figure 4. In that figure, object s is directly, but only partially, contained and shared by the three objects q, r and t. On the other hand, t is directly and uniquely contained by r. Also, s is indirectly (via t) partially contained by r. The graph on the right hand side of Figure 4 captures the geometric relationships between objects q, r, s and t, where a dashed (not dotted) arrow denotes direct shared containment and a solid arrow denotes direct unique containment.

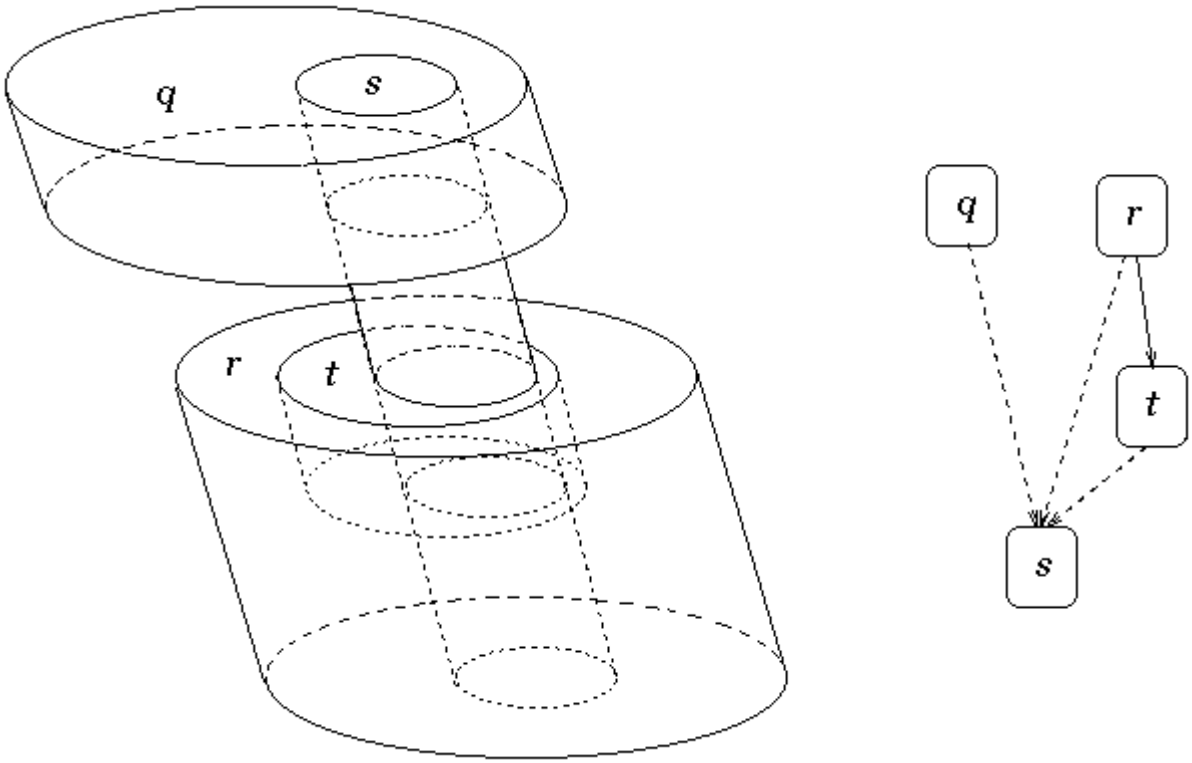


Figure 4: The geometry of shared containment

Figure 4 suggests three ideas implicit in the notion of shared and unique contain-

ment:

(1) an object cannot directly or indirectly contain itself, regardless of whether the containment is shared or unique;

(2) an object cannot be directly uniquely contained within two distinct objects (as in Section 2); and

(3) for any object, its set of directly contained but sharable objects is disjoint from its set of directly uniquely contained objects.

To capture these ideas formally, let

$$sdcon : \mathbb{O} \leftrightarrow \mathbb{O}$$

denote the relation of direct but sharable containment, i.e.

$$ob_1 \ \underline{sdcon} \ ob_2$$

if and only if object ob_1 directly contains but may share object ob_2. Let DC (direct containment) denote the relation

$$DC : \mathbb{O} \leftrightarrow \mathbb{O}$$

defined to be the union of the two relations $dcon$ (as defined in Section 2) and $sdcon$, i.e.

$$DC = dcon \cup sdcon.$$

The three conditions above respectively require that

$$\nexists ob : \mathbb{O} \bullet ob \ \underline{DC^+} \ ob,$$
$$dcon^\sim \in \mathbb{O} \nrightarrow \mathbb{O},$$
$$dcon \cap sdcon = \varnothing.$$

The notion of shared containment can be incorporated into Object-Z in much the same way that unique containment was modelled in Section 3. Briefly, let every class have two implicit attributes

$$sdcontain, scontain : \mathbb{P}\,\mathbb{O}$$

where $sdcontain$ denotes the set of directly contained but sharable objects, while $scontain$ denotes the directly and indirectly contained but sharable objects. Each class has an implicit invariant

$$\begin{aligned} scontain = \{ & ob : \mathbb{O} \mid \\ & ob \notin contain \\ & \exists\, s : \mathrm{seq}_1\, \mathbb{O} \bullet \\ & \qquad s(1) \in (dcontain \cup sdcontain) \\ & \qquad s(\#s) = ob \\ & \qquad \forall\, i : 1 \,..\, \#s - 1 \bullet s(i+1) \in (s(i).dcontain \cup s(i).sdcontain)\} \end{aligned}$$

In terms of the attributes *dcontain*, *contain*, *sdcontain* and *scontain*, the above conditions on the relations *dcon* and *sdcon* imply the following predicates are implicit invariants of any system:

$$\nexists ob : \mathbb{O} \bullet ob \in (ob.contain \cup ob.scontain),$$
$$\forall o_1, o_2 : \mathbb{O} \bullet o_1 \neq o_2 \Rightarrow o_1.dcontain \cap o_2.dcontain = \varnothing,$$
$$\forall ob : \mathbb{O} \bullet ob.dcontain \cap ob.sdcontain = \varnothing.$$

6.1 Example: Walls, Rooms and Buildings

Consider once again a campus consisting of buildings and rooms where each room in a building has a set of walls each of which may be shared with some other room. Furthermore, suppose that in the campus the buildings are physically separated so that no wall is shared between different buildings. An Object-Z specification of the campus would include the classes

Wall

...

Room

$ws : \mathbb{P}\, Wall_{\textcircled{S}}$

...

Building

$rs : \mathbb{P}\, Room_{\textcircled{C}}$
$ws : \mathbb{P}\, Wall_{\textcircled{C}}$

$\bigcup\{r : rs \bullet r.ws\} \subseteq ws$

...

where the notation '$_{\textcircled{S}}$', like the notation '$_{\textcircled{C}}$' introduced in Section 3.2, is a syntactic simplification identifying the directly contained but sharable objects. Without this simplification the state schema of the *Room* class would have been

$ws : \mathbb{P}\, Wall$

$sdcontain = ws$

This specification captures explicitly the geometric view that a building uniquely contains both its rooms and its walls, even although these walls may be shared between the rooms. Notice that this specification does not demand that each wall be shared between rooms; rather, it simply indicates that each wall is possibly shared (i.e. is sharable).

6.2 Example: Buildings, Streets and Suburbs

It is possible for an object within a system to both uniquely contain some objects and sharably contain others. For instance, consider a town consisting of suburbs, streets and buildings. An Object-Z specification of the town would include the classes

Building

...

Street

...

Suburb

$bs : \mathbb{P}\, Building_{\textcircled{C}}$
$ss : \mathbb{P}\, Street_{\textcircled{S}}$

...

This specification captures explicitly the geometric view that building are unique contained within suburbs, whereas streets may be contained but shared between suburbs.

6.3 Other Containment Geometries

Although the geometric notion of unique containment, and to a lesser extent that of shared containment, captures, in our experience, the geometry most commonly occurring in real systems, other geometries of object containment are possible. For example, consider the situation illustrated in Figure 5 where object m partially contains object n while at the same time n partially contains m.

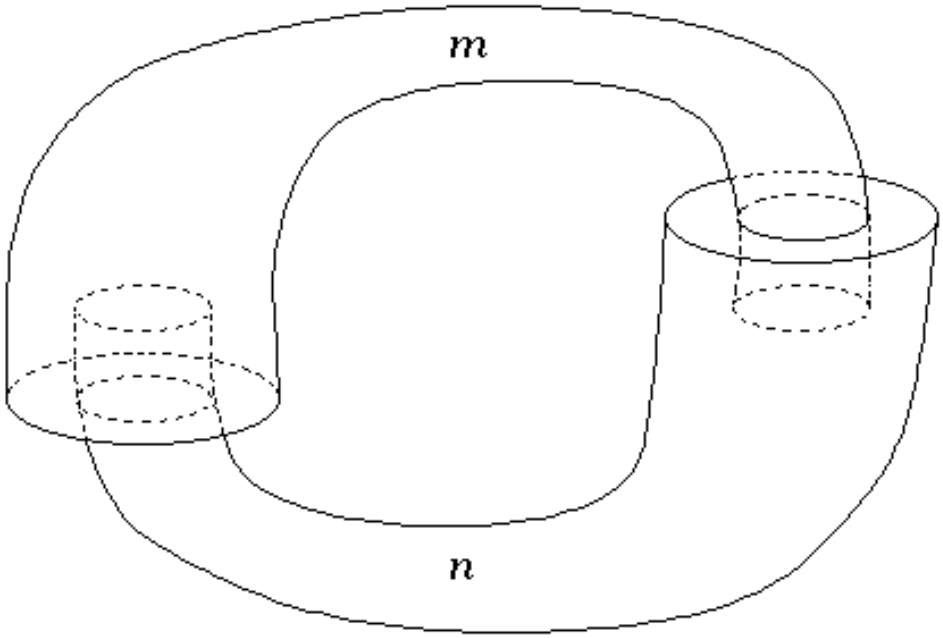


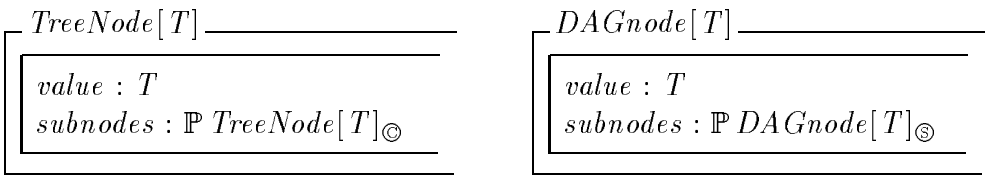
Figure 5: The geometry of circular partial containment

Although it would be possible to introduce specific notation to formally capture such geometries in Object-Z, as such structures only occur quite rarely in practice it is adequate to capture the properties implied by such geometries explicitly as class invariants (as in Section 2) when the need arises. An example is given in Section 7, when modelling the abstract structure of a doubly-linked list.

7 Containment in Abstract Structures

The object references that exist in object-oriented models of abstract recursive structures such as trees or directed acyclic graphs (DAGs) often satisfy the combinatorial properties of object containment, and hence the geometry of object containment can be applied directly when constructing object-oriented models for such structures. As

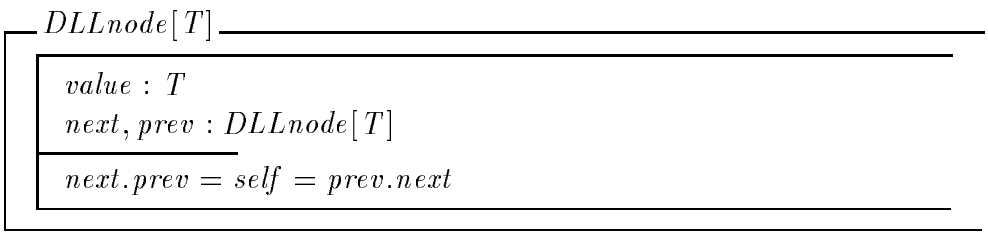
an illustration, consider the following (partial) Object-Z specifications of a tree node and a DAG node where T is a generic type.



Each node in the tree (or the DAG) has associated with it a value of type T and a (possibly empty) set of subnodes.

The use of object containment in the above examples guarantees a tree (or a DAG) structure without the need to state explicitly as invariants the properties of such a structure[8].

The notions of unique and shared containment are particularly suitable for modelling acyclic abstract structures, but inadequate for cyclic structures. For instance, an Object-Z specification of a node in a doubly-linked list would include



where the doubly-linked property is captured by an explicit invariant. As an example, consider two instances of the $DLLnode[T]$ class linked together, i.e. where each one is the *next* (and *prev*) of the other. The (circular) object references between the two instances (nodes) can be viewed as an abstract realisation of the geometry illustrated in Figure 5.

8 Conclusions

In this paper, the notion of object containment was captured within a formal framework, first by predicates incorporating the properties of containment within class invariants, and then by extending the Object-Z notation to capture the geometry of object containment directly. The advantage of this extension is that the properties of containment follow implicitly and do not need to be stated explicitly by invariants. Within this formal framework

- reference semantics for objects is sufficient: the introduction of value semantics (e.g. the *expanded* class type of Eiffel) is not needed to capture containment;
- there is a clear distinction between object containment and object control: it is possible to model systems where messages are sent to a contained object by objects other than the containing object; and

[8]The specifications of similar abstract structures but without using the containment notion can be found in [DD94, DR93].

- relocation of contained objects is possible without compromising the underlying geometry: the structure of containment is maintained implicitly.

The notion of object containment is particularly useful not only to explicitly capture geometric ideas of object location, but also to specify abstract acyclic structures.

Possible future research will be to use the presented formal framework to capture other important common object associations (see [KR94]).

Acknowledgements

We would like to thank Steven Atkinson, Alena Griffiths, Wendy Johnston, Andreas Prinz, Gordon Rose and Udaya Shankar for many helpful discussions on the issues raised in this paper. We also wish to thank the anonymous referees for many helpful suggestions.

References

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Addison-Wesley, 1991.

[Civ93] F. Civello. Role for composite objects in object-oriented analysis and design. In *Proc. 8th ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'93)*, 1993.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice-Hall, 1991.

[dCLF93] D. de Champeaux, D. Lea, and P. Faure. *Object-Oriented System Development.* Addison-Wesley, 1993.

[DD93] J. Dong and R. Duke. Class Union and Polymorphism. In C. Mingins, W. Haebich, J. Potter, and B. Meyer, editors, *Proc. 12th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS 12 & 9*, pages 181–190. Prentice-Hall, November 1993.

[DD94] J. Dong and R. Duke. An Object-Oriented Approach to the Formal Specification of ODP Trader. In J. Meer, B. Mahr, and S. Storp, editors, *Open Distributed Processing, II (ICODP'93)*, pages 341–352, Berlin, 1994. North-Holland.

[DKRS91] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice-Hall, 1991.

[DR93] R. Duke and G. Rose. Modelling object identity. In *Proc. 16th Australian Comput. Sci. Conf. (ACSC-16)*, pages 93–100, February 1993.

[DT93] T. Dillon and P.L. Tan. *Object-Oriented Conceptual Modeling.* Prentice-Hall, 1993.

[GR89] A. Goldberg and D. Robson. *Smalltalk 80: The Language.* Addison-Wesley, 1989.

[Hog91] J. Hogg. Islands: Aliasing Protection In Object-Oriented Languages. In *Proc. 6th ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91)*, 1991.

[HS92] B. Henderson-Sellers. *A Book of Object-Oriented Knowledge.* Object Oriented Series. Prentice-Hall, 1992.

[KR94] H. Kilov and J. Ross. *Information Modeling: An Object-Oriented Approach.* Object Oriented Series. Prentice-Hall, 1994.

[Lif93] R.H. Liffers. Inheritance versus Containment. *ACM SIGPLAN Notices*, 28(9):36–38, 1993.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* International Series in Computer Science. Prentice-Hall, 1988.

[Mey92] B. Meyer. *Eiffel: The Languages.* Prentice-Hall, 1992.

[Nie93] O. Nierstrasz. Composing Active Objects — The Next 700 Concurrent Object-Oriented Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 151–171. MIT Press, 1993.

[Ode92] J. Odell. Managing object complexity, part II: composition. *Journal of Object-Oriented Programming*, 5(6):17–20, 1992.

[Ros92] G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, 1992.

[Smi91] A. Smith. On recursive free types in Z. In *Proceedings of the Sixth Annual Z-User Meeting*, University of York, December 1991.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

Appendix: Glossary of Notation

Sets, Functions and Relations

$\mathbb{N}$	The set of natural numbers, i.e. $\{0, 1, 2, ...\}$
$\varnothing$	The empty set
$\mathbb{P}\, X$	Powerset: the set of all subsets of X
$\#X$	Size (number of members) of a finite set
$\{D \bullet t\}$	The set of values of the term t for the variables declared in D, e.g. $\{n : \mathbb{N} \bullet 2 * n\}$ is the set of even natural numbers
$\bigcup S$	Distributed union of a set of sets, e.g. if $S \in \mathbb{P}\mathbb{P}\, X$ then $\bigcup S = \{x : X \mid \exists\, s : S \bullet x \in s\}$
$X \times Y$	Cartesian product: the set of ordered pairs
$X \leftrightarrow Y$	The set of all relations from X to Y; a relation from X to Y is a subset of $X \times Y$
$x \underline{R}\ y$	x is related by the relation R to y, i.e. $(x, y) \in R$
$X \nrightarrow Y$	The set of partial functions from X to Y; the domain of the function is not necessarily all of X
R^+	The transitive closure of relation R, i.e. a pair (x_1, x_n) is in the relation R^+ if and only if there exists a finite sequence $x_1, x_2, ..., x_n$, where $n \geq 2$, such that $(x_1, x_2) \in R$, $(x_2, x_3) \in R$, ... and $(x_{n-1}, x_n) \in R$

Object-Z Overview

Object-Z is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring.

Classes

A class is a template for *objects* of that class: for each such object, its states are instances of the class' state schema and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class.

Syntactically, a class definition is a named box, optionally with generic parameters. In this box the constituents of the class are defined and related. The main constituents are: a state schema, an initial state schema and operation schemas.

ClassName[*generic parameters*]

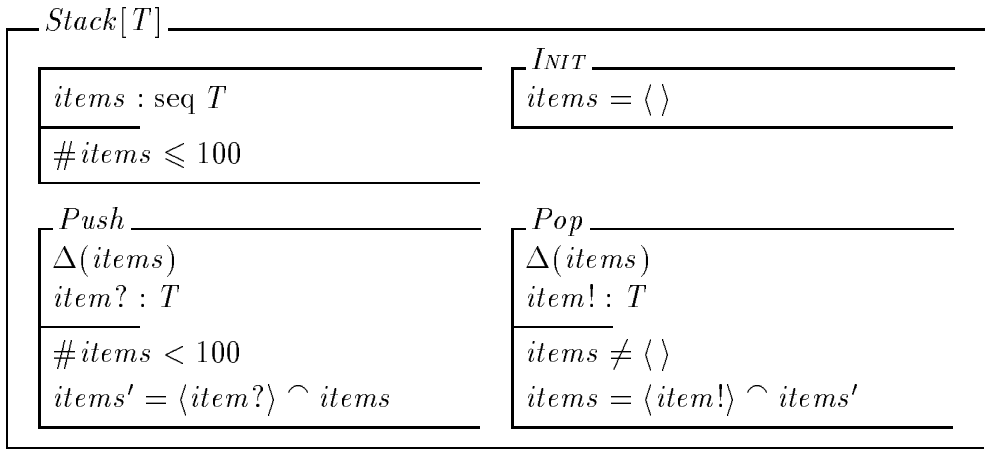
state schema

initial state schema

operation schemas

A generic stack example

Consider the following specification of the generic class *Stack*.



The state schema is nameless and contains declarations (the attributes) above the short dividing line and a predicate (class invariant) below the line. In this example, it has one attribute *items* denoting a sequence of elements of the generic type T. The class invariant stipulates that the size of the sequence cannot exceed 100.

The initial state schema is distinguished by the keyword *INIT*. The state schema is implicitly included in the initial state schema. In this example, an initialised stack contains no elements of T (i.e. *items* is the empty sequence).

The remaining two schemas are operation schemas. Operation schemas have a Δ-list of those attributes whose values may change. By convention, no Δ-list means no attribute changes value. Every operation schema implicitly includes the state schema in un-primed form (the state before the operation) and primed form (the state after the operation). Hence the class invariant holds at all times: in each possible initial state and before and after each operation.

In this example, operation *Push* prepends a given input *item*? to the existing sequence of items provided the stack has not already reached its maximum size (an identifier ending in '?' denotes an input). Operation *Pop* outputs a value *item*! defined as the head of sequence *items* and reduces *items* to the tail of its original value (an identifier ending in '!' denotes an output).

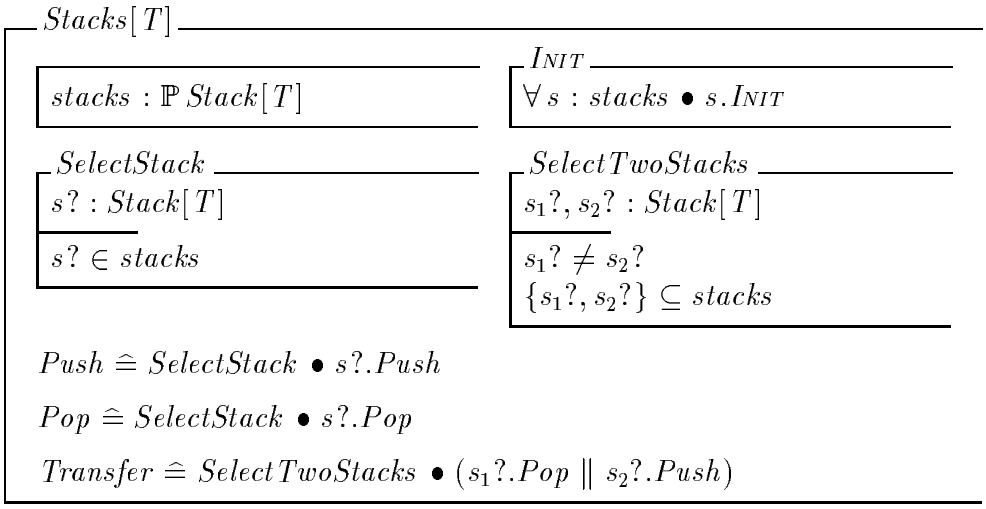
Instantiation

Objects may have object references as attributes, i.e. conceptually, an object may have constituent objects. Such references may either be individually named or occur in aggregates. For example, the declaration $c : C$ declares c to be a reference to an object of the class described by C. A declaration $c, d : C$ need not mean that c and d reference distinct objects. If the intention is that they do so at all times, then the predicate $c \neq d$ would be included in the class invariant.

The term $c.att$ denotes the value of attribute *att* of the object referenced by c, and $c.Op$ denotes the evolution of the object according to the definition of *Op* in the class C.

Stack Aggregation Example

Suppose we want to model an aggregate of stacks with operations to push an item onto, or pop an item off, any stack in the aggregation, and to transfer an item from one stack to another.



The declaration $stacks : \mathbb{P}\, Stack[T]$ models an aggregate of stack objects.

Evolution of a constituent stack object in the aggregate is effected by defining a selection environment such as *SelectStack* that selects a stack; the operations *Push* and *Pop* are then applied to the chosen object. (The notation $schema_1 \bullet schema_2$ means that variables declared in the signature of $schema_1$ are accessible when interpreting $schema_2$.)

Two stack objects of the aggregate may be selected by *SelectTwoStacks* to undergo an evolutionary step concurrently. The parallel operator, '‖', used in the definition of *Transfer* achieves inter-object communication: the operator conjoins operation schemas but also identifies (equates) and hides inputs and outputs having the same type and basename (i.e. apart from '?' or '!').

Further details on Object-Z are given in [DKRS91] and [Ros92].