VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

Towards a Model of Encapsulation

James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, and Dave Clarke

Technical Report CS-TR-03/6
April 2003

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Towards a Model of Encapsulation

James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, and Dave Clarke

Technical Report CS-TR-03/6
April 2003

Abstract

Encapsulation is a founding principle of object-oriented programming: to this end, there have been a number recent of proposals to increase programming languages support for encapsulation. While many of these proposals are similar in concept, it is often difficult to describe their effects in practice, or to evaluate clearly how related proposals differ from each other. We are developing a general topological model of encapsulation for object-oriented languages, based a program's object graph. Using this model, we can characterise a range of confinement, ownership, and alias protection schemes, in terms of their underlying encapsulation function.

This analysis should help programmers understand the encapsulation provided by programming languages, assist students to better compare and contrast the features of different languages, and help language designers to craft the encapsulation schemes of forthcoming programming languages.

Publishing Information

Workshop at the European Conference for Object-Oriented Programming, 2003

Author Information

James Noble, Robert Biddle, and Alex Potanin are based at Victoria University of Wellington, Ewan Tempero is based at The University of Auckland, and Dave Clarke is based at Institute of Information and Computing Sciences, Utrecht University

Towards a Model of Encapsulation

James Noble[1], Robert Biddle[1], Ewan Tempero[2], Alex Potanin[1], and Dave Clarke[3]

[1] School of Mathematical and Computing Sciences, Victoria University of Wellington
[2] Department of Computer Science, The University of Auckland
[3] Institute of Information and Computing Sciences, Utrecht University

Abstract. Encapsulation is a founding principle of object-oriented programming: to this end, there have been a number recent of proposals to increase programming languages support for encapsulation. While many of these proposals are similar in concept, it is often difficult to describe their effects in practice, or to evaluate clearly how related proposals differ from each other. We are developing a general topological model of encapsulation for object-oriented languages, based a program's object graph. Using this model, we can characterise a range of confinement, ownership, and alias protection schemes, in terms of their underlying encapsulation function. This analysis should help programmers understand the encapsulation provided by programming languages, assist students to better compare and contrast the features of different languages, and help language designers to craft the encapsulation schemes of forthcoming programming languages.

1 Introduction

We are developing a simple topological model of encapsulation in object-oriented systems, programming languages, and more general computer systems. This model has three parts — an *access graph* describing the topology of the system under consideration; an *encapsulation function* describing how particular parts of the system are encapsulated; and an *encapsulation constraint* that, when true, captures the fact that the encapsulation function partitions the access graph so that so part of it is actually encapsulated.

For this position paper, we take a statement of Dan Ingalls as the defining quality of encapsulation[1]:

> *No component in a complex system should depend on the internal details of any other component.* [13]

As far as we are aware, there is as yet no agreed formalisation of a definition of encapsulation: our model is intended to be a step in that direction.

1.1 Access Graph

The first part of our model is an *access graph*. An access graph models the topology of a system: nodes ($\mathcal{N}$) in the graph represent individual objects in the model, and edges ($\mathcal{E}$) between nodes represent accesses between objects.

[1] Glossing over the fact that Ingalls was defining *modularity* rather than *encapsulation*.

Our formulation of encapsulation does depend upon whether the graph is rooted or directed: if an access graph has a distinguished root node we can incorporate it easily but we do not require it; while in an undirected graph we take an edge (a, b) to mean that a accesses b and b accesses a.

Some notation: we write $\mathcal{S}$ for all the objects in the system, that is, the set of all nodes in the graph; $\{a\}\lhd$ to be the set of all nodes that have edges beginning from a and $\rhd\{a\}$ to be the set of all nodes that have edges ending at a; $\{a\}\lll$ and $\ggg\{a\}$ for their transitive closures (all nodes which can be reached from a and which can reach a respectively); $a \longrightarrow b$ to mean that there is an edge from a to b; *paths*(a, b) for the set of all paths (sequences of nodes) from node a to node b; *edges*(a, b) for the set of all edge labels along *paths*(a, b); $a \sqsubseteq b$ to mean that a is a dominator for b, that is, every path to b leads through a. Nodes and edges may (or may not) be labelled: node labels are typically given by boolean predicates (as functions from $\mathcal{N}$ to $\mathbb{B}$); or functions to (sets of) nodes, components, or model-specific sets; edges may be labeled simply (e.g. i, r, w) and graph operations subscripted to restrict to matching edges (i.e. $a\lll_i$ is all nodes reachable from a along edges labelled i).

Our access graph is unremarkable in its suburban monotony, being at heart a directed graph: our whole model of encapsulation is similarly simple-minded. The reason for this naïve formulation is that the access graph is itself an abstraction: our model of encapsulation will work over *any* other directed graph model that meets these criteria. In this position paper, we will consider only *object graphs*, with objects as nodes and their variables (references between objects) as edges. We may build this graph informally based on some kind of object identifiers, use Zeller's Memory Graphs [17] or Hoare and Jifeng's trace model [11], or some other formulation of an object-oriented program that also is based on a directed graph [9]. We can decorate the graph as necessary to distinguish between static or dynamic object accesses, the classes, packages, modules, block structure, or files to which objects belong. We expect the ubiquity of the access graph will allow us to address some other kinds of encapsulation in object-oriented systems and encapsulation in non-object-oriented (and non-informatic) systems.

1.2 Encapsulation Function

The second part of the model is the encapsulation function. The encapsulation function partitions the system ($\mathcal{S}$) by taking a identifier of an *encapsulated component* ($\mathcal{C}$) to a three-tuple of sets of access graph nodes that respectively describe the encapsulation *boundary* ($\mathcal{B}$), the *inside* ($\mathcal{I}$), and the *external* references ($\mathcal{R}$) of the component.

$$\begin{aligned} &\mathcal{C} : \mathrm{id} \\ &\mathcal{S}, \mathcal{B}, \mathcal{I}, \mathcal{R}, \mathcal{O} : \mathbb{P}\mathcal{N} \end{aligned}$$

$$\mathbf{e}(\mathcal{C}) = \begin{pmatrix} \mathcal{B} = \mathcal{B} \\ \mathcal{I} = \mathcal{I} \\ \mathcal{R} = \mathcal{R} \end{pmatrix}$$

The boundary of a component represents the interface that components presents to the rest of system, while the inside is that part of the component which is encapsulated, and may itself be a network of interconnected objects. The inside of a component can only be accessed via the components interface, that is, by crossing its encapsulation boundary. The external references of a component are nodes which the boundary or inside accesses directly, but which are not contained within boundary or inside of the component. Finally, the *outside* ($\mathcal{O}$) of a component are all nodes which are neither inside the component nor on its boundary — including the component's external references.

$$\mathcal{O} = \mathcal{S} - (\mathcal{B} \cup \mathcal{I})$$
$$\mathcal{R} \subseteq \mathcal{O}$$

This is illustrated in figure 1 below. All the nodes in the graph $\mathcal{S}$ represent the whole system: those nodes outside any subset boundary are in the outside $\mathcal{O}$ of the component shown in the diagram.. The boundary $\mathcal{B}$ (the left subset) provide the interface to the encapsulated component: they can be accessed from anywhere — outside the component, inside it, from other boundary nodes, or from the component's external references. The centre subset in the diagram represent the inside $\mathcal{I}$ of the component: nodes here may only be accessed by each other or by the boundary nodes. The right-most subset are the outside nodes referred to by the inside and boundary of the component — its external references $\mathcal{R}$.

Note that we make a distinction between components ($\mathcal{C}$) and nodes ($\mathcal{N}$): components are not nodes: rather a component *encapsulates* several nodes (via its encapsulation function $\mathbf{e}$) by identifying an encapsulation boundary, the nodes inside that boundary, and any external references leaving that boundary. In an object graph, objects are nodes, however compoments may be based upon a range of structures, including objects but also packages, classes, and universes, as we will see below.

The argument to the encapsulation function — a component identifier — effectively chooses a particular partition of the system. For that component, the function selects those nodes that will be on the boundary, inside, external, and outside. The interpretation of the component identifier depends upon the particular scheme being modelled, and is typically linked to the access graph proper via node labels. For example, for package-based encapsulation (such as confined types, see section 2.6 below) the unit of encapsulation is a package, so component identifiers model packages and nodes need a labelling function (say *package* : $\mathcal{N} \rightarrow \mathcal{C}$) to identify the package to which they belong. An encapsulation function for such a scheme takes an identifier (such as "`java.lang.util`") as its argument to select the package that is encapsulated: the function itself will then choose objects based on their package (e.g. grouping objects of classes in `java.lang.util`). By contrast, in object-based encapsulation schemes such as islands or balloons, each node forms an encapsulated component, so we let $\mathcal{C} = \mathcal{N}$.

1.3 Encapsulation Constraints

So far, we have an access graph as a (directed) graph; and an encapsulation function which selects nodes from that graph. The third part of our model is the *encapsulation*

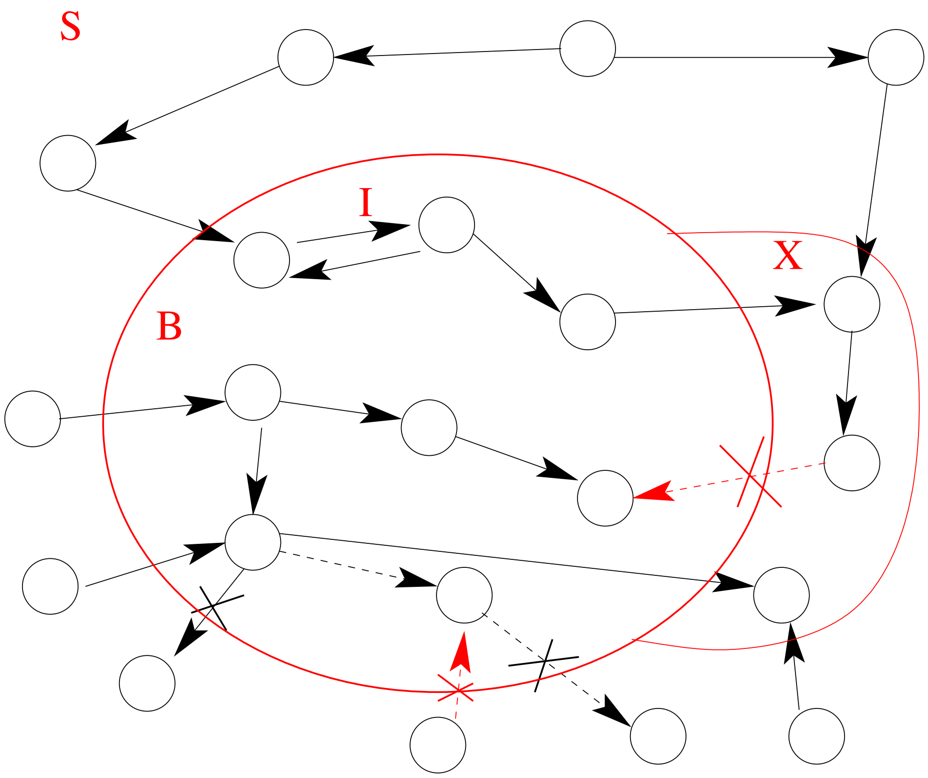

Fig. 1. An encapsulated component

constraints that link the access graph and encapsulation function. These constraints ensure that the encapsulation function does actually describe encapsulation: that inside nodes are only accessed via the boundary, and that outside nodes are only accessed via the external references.

More formally, the encapsulation constraints are as follows: first, a component's boundary, inside, and externals must be disjoint (this implies that the externals must be outside the component).

$$\mathcal{B} \cap \mathcal{I} = \mathcal{B} \cap \mathcal{R} = \mathcal{I} \cap \mathcal{R} = \{\}$$

The second constraint is the most important one: the boundary must actually be a boundary for the inside. To be a valid encapsuation the nodes inside an encapsulated component can only be accessed via the component's boundary: that is, any edges ending at inside nodes can only come from other inside nodes or boundary nodes:

$$\triangleright\mathcal{I} \subseteq (\mathcal{B} \cup \mathcal{I})$$

Alternatively we could state that all paths from the outside of a component into the inside must pass through the boundary.

$$\forall o \in \mathcal{O} : \forall i \in \mathcal{I} : \forall p \in \textit{paths}(o, i) : \exists b \in p \text{ s.t. } b \in \mathcal{B}$$

where *paths*(a, b) is all paths (as a set of sets of nodes) from i to o.

Finally we require that every outside node directly accessed by an inside node is recorded in the component's externals:

$$(\mathcal{B} \cup \mathcal{I}) \lhd \subseteq (\mathcal{B} \cup \mathcal{I} \cup \mathcal{R})$$

Note that this sets a lower bound on the extent of the external references — a component may include outside objects in its external references even if it does not refer to them. This is useful because it allows us to set $\mathcal{R} = \mathcal{O}$ to indicate that an object's external references are unconstrained, and in any event, one can always define an alternative encapsulation function with a tighter external reference set.

This formal definition is designed to capture the intent of Dan Ingalls' statement above:

> *No component in a complex system should depend on the internal details of any other component.* [13]

In our model, the "*external aspects*" of an encapsulated component lie on its boundary, while the *"internal details"* are its inside. The encapsulation constraints ensure that these internal details of a component are hidden from every external component in the system.

1.4 Nested Encapsulation

These definitions imply that encapsulated components may be nested inside each other. We say that c_1 contains c_2 ($c_1 \leq c_2$) or c_2 is inside c_1 ($c_2 \geq c_1$) if all c_2's boundary and inside are part of the boundary and inside of c_1. That is:

$$c_1 \leq c_2 \doteq \mathcal{B}_{c_2} \subseteq (\mathcal{B}_{c_1} \cup \mathcal{I}_{c_1}) \wedge \mathcal{I}_{c_2} \subseteq (\mathcal{I}_{c_1})$$

One consequence of this definition is that the external references of the inner component must be contained within the inside, boundary, or external references of the outer component:

$$\mathcal{R}_{c_2} \subseteq (\mathcal{B}_{c_1} \cup \mathcal{I}_{c_1} \cup \mathcal{R}_{c_1})$$

2 Encapsulation Schemes over Object Graphs

Having outlined our model of encapsulation, in this section we apply that model to a range of encapsulation schemes over object graphs, beginning with simple implicit models and progressing to more complex schemes. We describe each scheme by their characteristic encapsulation function.

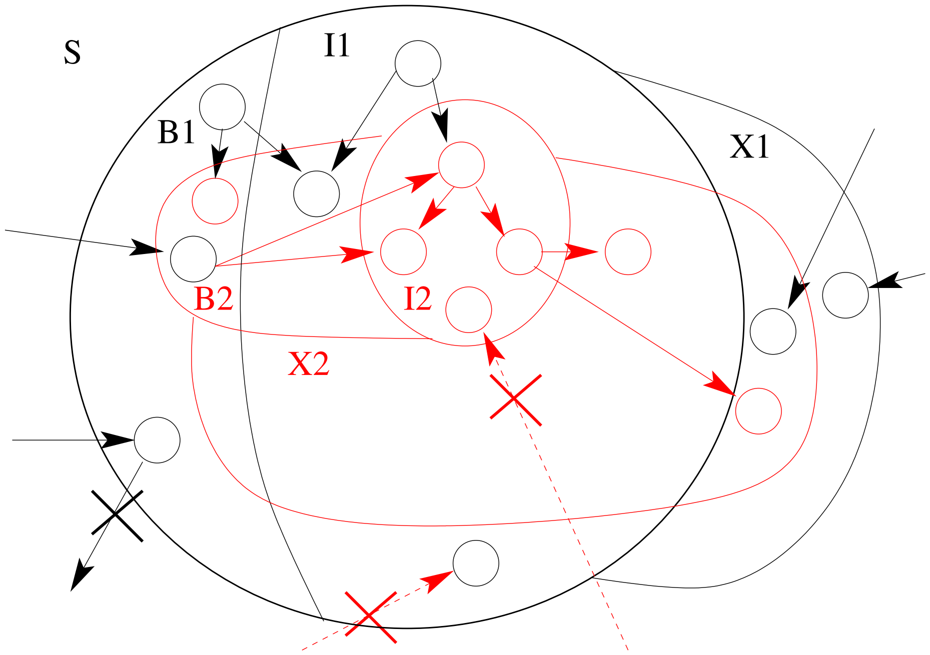

Fig. 2. Nested Components

An object graph is part of a program's operational state: the graph will evolve as the program runs, as objects are created and deleted and as assignments change internode references. This is the reason that we characterise encapsulation *schemes* by encapsulation *functions*: these functions are implicitly parameterised by the object graphs to which they are applied. Typically, an encapsulation scheme will mandate that the encapsulation constraints hold at all times in all runs of all programs to which the scheme applies. Different schemes will use a wide range of *mechanisms* to achieve this, from dynamic checks at assignment or invocation, through static annotations, extended type systems, abstract interpretation, theorem proving, or even providing no enforcement at all and simply relying on programmers' adherence to conventions. One key advantage of our model of encapsulation functions (and access graphs) is that they can abstract away considerations of mechanism and provide succinct characterisations of the encapsulation policies supported by each scheme — in particular, the topology of encapsulation that each scheme provides.

2.1 Islands: Full Encapsulation

Hogg's Islands [12] was the first alias protection scheme advocated for an object-oriented language. The key notion of an Island is that some classes in the program are distinguished as *bridge* classes: instances of those classes are subject to a series of syntactic restrictions on their parameter aliasing, so that all objects reachable from the bridge — i.e. the objects making up the island — are *only* reachable from the bridge.

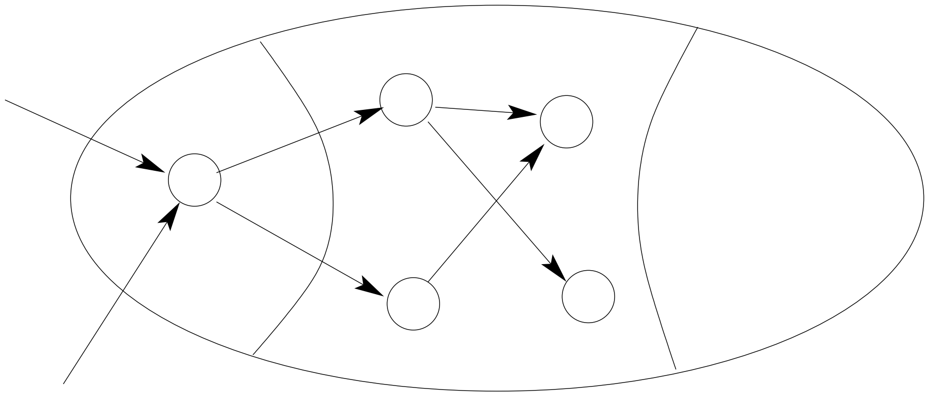

Fig. 3. Full Encapsulation

In terms of an encapsulation function, an object o that has been identified as an bridge (using the label $bridge : \mathcal{N} \rightarrow \mathbb{B}$) has itself as a boundary, all objects reachable via that boundary as its inside, and no external references:

$$\mathbf{e}_{island}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \\ \mathcal{I} = o \lll \\ \mathcal{R} = \{\,\} \end{pmatrix} \\ \text{where } bridge(o)$$

Islands are one example of an alias protection scheme with a *full encapsulation* topology: the encapsulation functions of such schemes are characterised by having empty external references. Banerjee and Naumann's heap confiement [3] also mandates full encapsulation, as does Almeida's Balloon Types [2].

2.2 Uniqueness

Islands also supports another common type of encapsulation, uniqueness. A classically unique object has only one incoming reference ($| \rhd u| = 1$) and so a unique object is always encapsulated by the sole object that refers to it. This condition produces a range of encapsulation function.

Most generally, a unique object may refer to any other object in the system:

$$\mathbf{e}_{unique}(u) = \begin{pmatrix} \mathcal{B} = \rhd u \\ \mathcal{I} = \{u\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix} \\ \text{where } | \rhd \{u\}| = 1$$

where we say that the external references are the outside ($\mathcal{O}$) of the unique object to mean that the references are unconstrained (not that the object necessarily refers to every outside object!). Note also that this definitions works by placing the unique object into the inside of the encapsulation: the encapsulation boundary is the (sole) object which refers to this object.

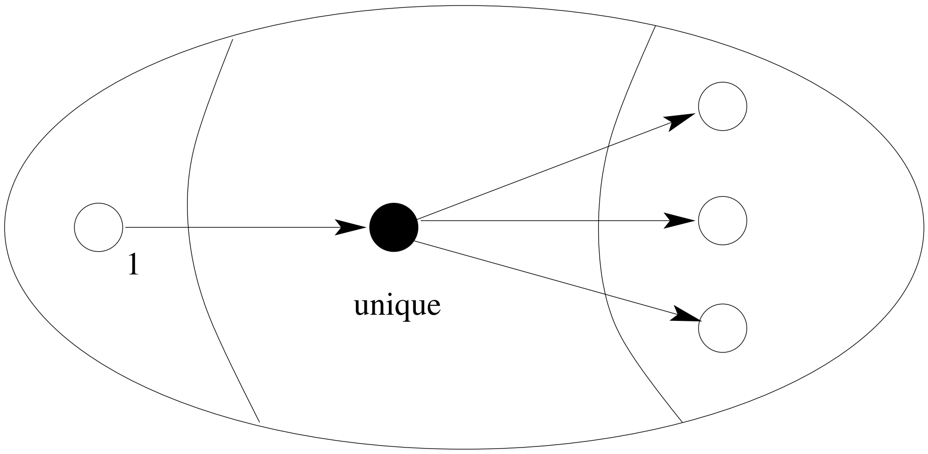

Fig. 4. A Unique Object

Clarke and Wrigstrad have recently demonstrated that a weaker formulation of uniqueness can provide all the benefits of full uniqueness but is significantly more flexible. An *externally unique* object [6, 7] may have only one reference from outside, but may have any number of other references from inside:

$$\mathbf{e}_{\textit{external-unique}}(u) = \begin{pmatrix} \mathcal{B} = \{v\} \\ \mathcal{I} = \{u\} \cup \{o | u \sqsubseteq o\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix}$$
$$\text{where } v = (\rhd\{u\} - \mathcal{I}) \text{ and } |v| = 1$$

Again, the encapsulation boundary is the object that refers to the externally unique object. Here, however, an unique object contains a number of other objects but any incoming references from these objects do *not* count against the uniqueness of the externally unique object — external uniqueness counts only those references from the boundary in to the unique object, not references from the inside of the object.

2.3 Balloon Types

Alemida's Balloon Types [2] provide full encapsulation, however, they are also constrained to be unique. Given that balloon objects are labeled ($\textit{balloon} : \mathcal{N} \to \mathbb{B}$), we can model this with two nested encapsulations, a inner full encapsulation

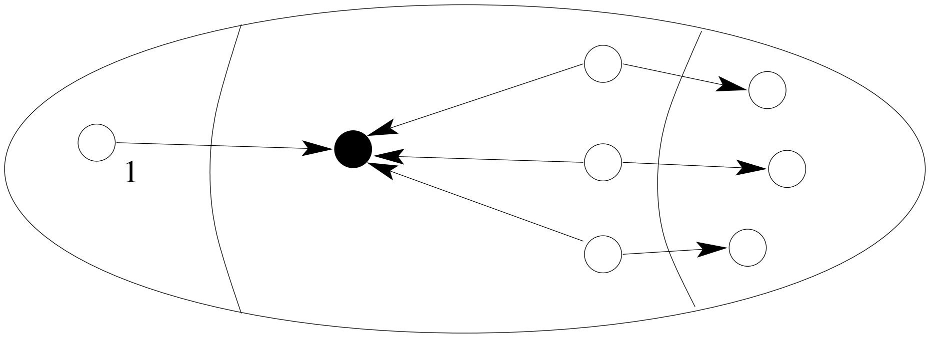

Fig. 5. External Uniqueness

$$\mathsf{e}_{\textit{inner balloon}}(b) = \begin{pmatrix} \mathcal{B} = \{b\} \\ \mathcal{I} = \{b\}\lll \\ \mathcal{R} = \{\,\} \end{pmatrix}$$
$$\text{where } \textit{balloon}(b)$$

and an outer unique object:

$$\mathsf{e}_{\textit{outer balloon}}(b) = \begin{pmatrix} \mathcal{B} = \rhd\{b\} \\ \mathcal{I} = \{b\} \\ \mathcal{R} = \{b\}\lll \end{pmatrix}$$
$$\text{where } \textit{balloon}(b) \wedge |\rhd\{b\}| = 1$$

or as a single encapsulation function encompassing both the balloon object, the contents of the balloon and having its boundary as the the (sharable) object holding the single reference to balloon object proper:

$$\mathsf{e}_{\textit{balloon}}(b) = \begin{pmatrix} \mathcal{B} = \rhd\{b\} \\ \mathcal{I} = \{b\} \cup \{b\}\lll \\ \mathcal{R} = \{\,\} \end{pmatrix}$$
$$\text{where } \textit{balloon}(b) \wedge |\rhd\{b\}| = 1$$

These are then nested, so that $\mathsf{e}_{\textit{balloon}} \leq \mathsf{e}_{\textit{inner-balloon}}$ and $\mathsf{e}_{\textit{balloon}} \leq \mathsf{e}_{\textit{outer-balloon}}$.

2.4 Flexible Alias Protection

Flexible Alias Protection [15] was proposed to resolve problems with full static alias protection schemes such as Islands and Balloons: these schemes were simultaneously too lax, in that dynamic references could penetrate Islands and plain balloons, and too strict, as no external references were permitted. Flexible Alias Protection divided the objects within an "alias-protected container" into two main categories: *representation*

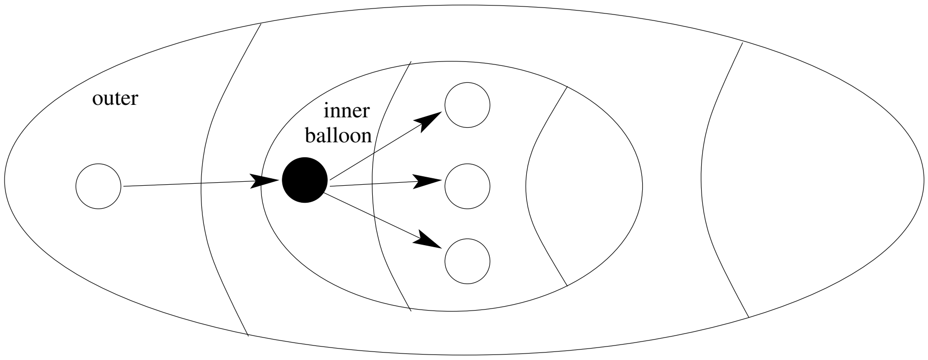

Fig. 6. Balloon Types

objects which were private, and the *arguments* objects which could be shared. Flexible alias protection used "modes" annotating static types to track which objects were representation and which arguments: we can model that where with a pair of labelling functions ($rep, arg : \mathcal{N} \rightarrow \mathbb{P}\mathcal{N}$). Unlike Islands, Flexible Alias Protection maintained the same encapsulation on dynamic access as static access — representation objects could never be accessed externally.

To a first approximation, the encapsulation function for Flexible Alias Protection is:

$$\mathbf{e}_{flexible}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \\ \mathcal{I} = rep(o) \\ \mathcal{R} = arg(o) \end{pmatrix}$$

with an object's representation being its inside, and arguments its external references. Flexible Alias Protection's restrictions on the use of modes (types of different mode are not assignment compatible; representation modes cannot appear in a protected container's interface, and so on) ensure that the labelling actually resulted in an encapsulation.

2.5 Ownership Types

Ownership Types [8] were first designed to formalise the topological restrictions of Flexible Alias Protection, although they have since found many difference applications. The key idea behind ownership types is that a type system statically enforces topological restrictions based on an ownership relation between objects: we can model this relationship by labelling the basic object graph so that every node has an owner ($owner : \mathcal{N} \rightarrow \mathcal{N}$). Owners are nested ($owner(i) \succeq i$), the ownership relation is constrained to form a convergent tree: ($owner(root) = root$) and the type system then enforces a containment invariant:

Owners are nested

$$s \longrightarrow t \Rightarrow s \preceq owner(t)$$

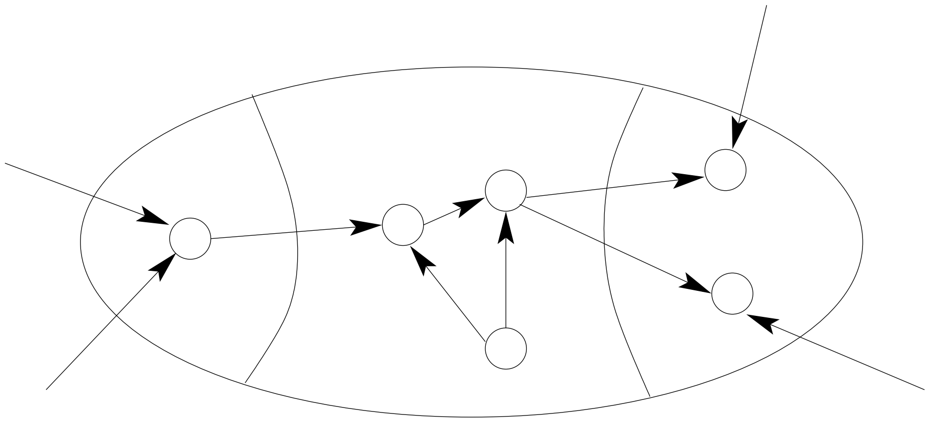

Fig. 7. Flexible Encapsulation Topology

that is, the source of an edge must be owned by the owner of the target of an edge. The containment invariant ensures that ownership partitions the graph into a series of nested encapsulations based on each object owning its inside.

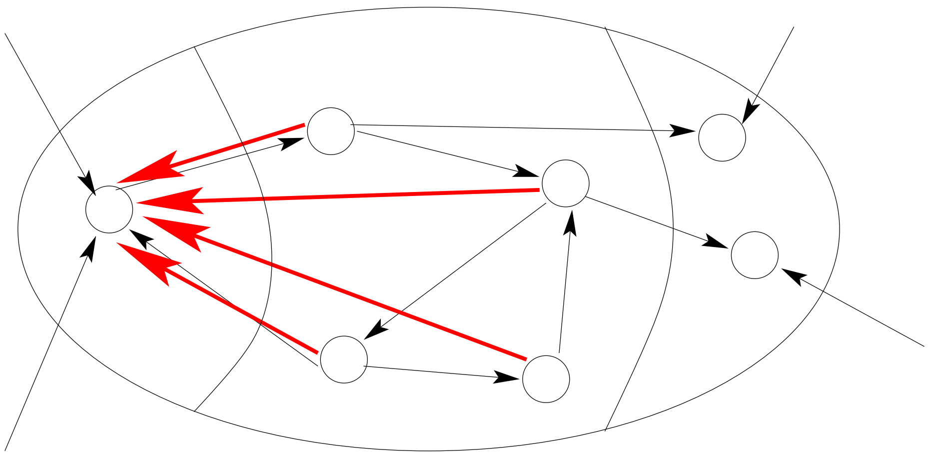

Fig. 8. Ownership Types

Using this *ownership graph* as the access graph, the most basic encapsulation function for ownership types have the form of:

$$\mathbf{e}_{\textit{simple-ownership}}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \\ \mathcal{I} = \{i \mid owner(i) = o\} \\ \mathcal{R} = \{r \mid o \preceq owner(r)\} \end{pmatrix}$$

The external reference expression for ownership types can be simplified if (the types of) objects are explicitly parameterised with the ownership of the objects to which they may refer ($params : \mathcal{N} - > \mathbb{P}\mathcal{N}$):

$$\mathbf{e}_{\textit{param-ownership}}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \\ \mathcal{I} = \{i \mid owner(i) = o\} \\ \mathcal{R} = \{r \mid owner(r) \in params(o)\} \end{pmatrix}$$
$$\text{where } \forall p \in params(o) : o \preceq p$$

These ownership type systems are known as *owners-as-dominators* models, because the containment invariant ensures that every object is dominated by their sole owner ($owner(o) \sqsubseteq o$). The encapsulation function shows how this forms an encapsulation: in fact the tree of owners gives a matching tree of nested encapsulations:

$$o_1 \succeq o_2 \Rightarrow \mathbf{e}(o_1) \leq \mathbf{e}(o_2)$$

Because the owners-as-dominators versions of ownership types limits a component's boundary to be a single object, these simple ownership types restrict some programming idioms. Recently, Boyapati and Liskov have extended ownership types to allow multiple objects in a component's boundary, where the extra objects are Java (or BETA) style inner objects nested within the main object [5]. This changes the encapsulation function as follows:

$$\mathbf{e}_{\textit{inner-ownership}}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \cup \{b \mid inner(b) = o\} \\ \mathcal{I} = \{i \mid owner(i) = o\} \\ \mathcal{R} = \{r \mid o \preceq owner(r)\} \end{pmatrix}$$

where $inner : \mathcal{N} \rightarrow \mathcal{N}$ gives the object within which a (non-static) inner class instsance is nested.

2.6 Other Type-based schemes

There are a range of other type-based encapsulation schemes. Confined types [4, 10] are a package-wide type discipline: some types are marked as confined, and instances of these classes can only be accessed by instances from the same package. Objects that are not confined within the package provide the interface by which they confined types are manipulated. We model this by decorating every object with the package to which their class belongs ($package : \mathcal{N} \rightarrow \mathcal{C}$) and a bit to indicate whether or not that class is confined[2] ($confined : \mathcal{N} \rightarrow \mathbb{B}$):

[2] We could add another layer of indirection here to treat classes explicitly, but this would needlessly complicate the model.

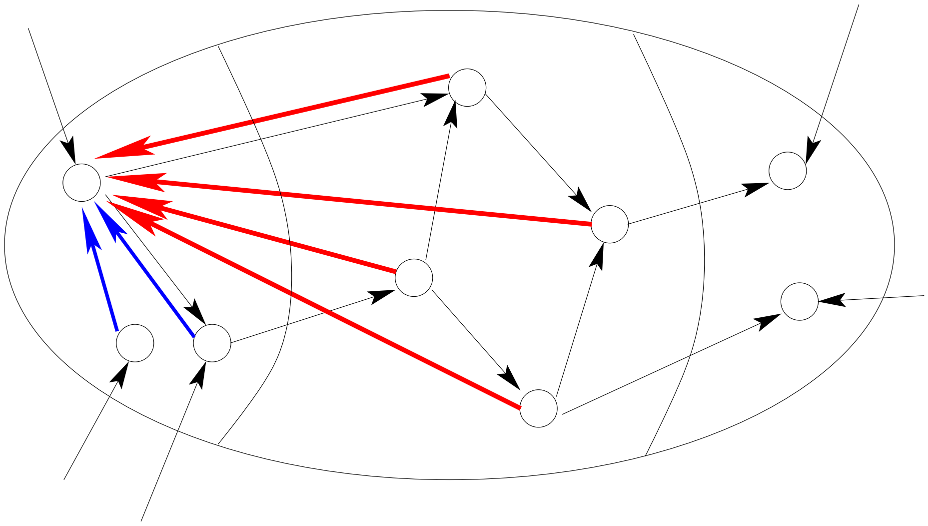

Fig. 9. Extended Ownership Types

$$\mathbf{e}_{\mathit{confined}}(p) = \begin{pmatrix} \mathcal{B} = \{o \mid \mathit{package}(o) = p \wedge \neg \mathit{confined}(o)\} \\ \mathcal{I} = \{o \mid \mathit{package}(o) = p \wedge \mathit{confined}(o)\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix}$$

Universes [14] are an alternative to ownership types, based on read-only references rather than parameterisation. In particular, objects are encapsulated only with respect to read-write references: read-only references are unencapsulated. A universe plays a similar role to an owner in ownership types (every object belongs to a universe — $\mathit{universe} : \mathcal{N} \rightarrow \mathcal{C}$), however as well as encapsulation based on objects (called object universes), universes also supports wider encapsulation (called type universes).

Considering only read-write references, then, every object has one object universe ($\mathit{objectu} : \mathcal{N} \rightarrow \mathcal{C}$, one-to-one) but may be associated with a set of type universes ($\mathit{typeu} : \mathcal{N} \rightarrow \mathbb{P}\mathcal{C}$). An object that uses only its own object universe is fully encapsulated:

$$\mathbf{e}_{\mathit{object\text{-}universe}}(u) = \begin{pmatrix} \mathcal{B} = \{o \mid \mathit{objectu}(o) = u\} \\ \mathcal{I} = \{i \mid \mathit{universe}(i) = u\} \\ \mathcal{R} = \{\,\} \end{pmatrix}$$
where u is the unique object universe of o

Alternatively, we can consider the encapsulation afforded by a type universe:

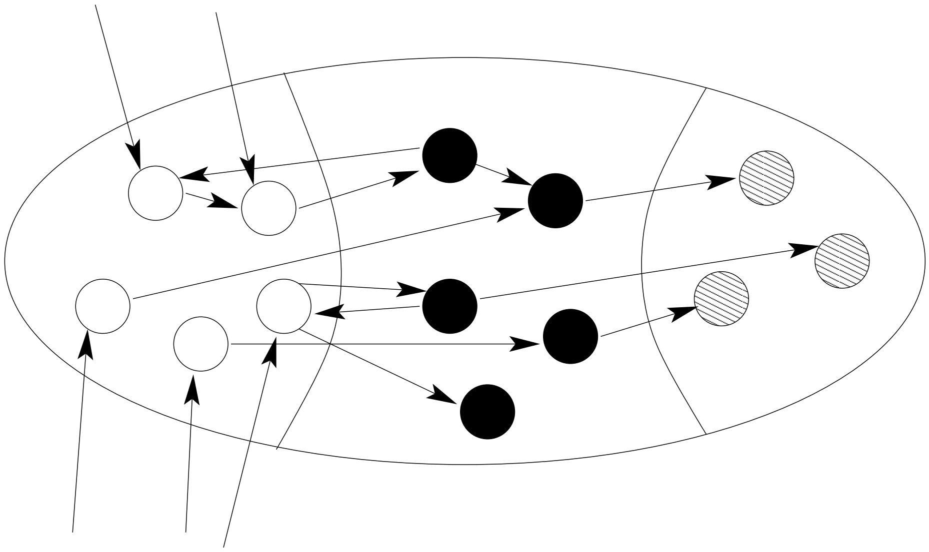

Fig. 10. Confined Types

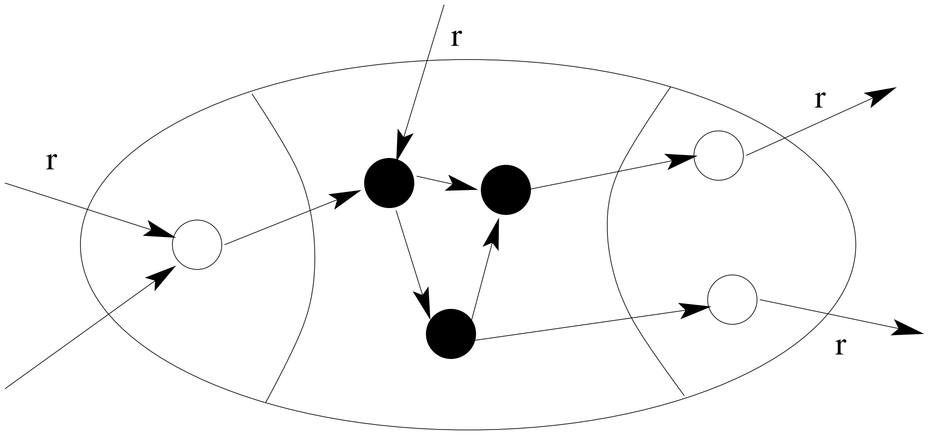

Fig. 11. Universes

$$\mathbf{e}_{\textit{type-universe}}(u) = \begin{pmatrix} \mathcal{B} = \{o \mid \textit{typeu}(o) \ni u\} \\ \mathcal{I} = \{i \mid \textit{universe}(i) = u\} \\ \mathcal{R} = \{\,\} \end{pmatrix}$$
where u is a type universe

3 Conclusion

In this position paper we have outlined our generic model of encapsulation, and applied that to a number of alias protection schemes over object graphs. In the future, we plan to apply this model to more object-graph schemes [1, 16]; to finer grained access graphs for programming languages; and to encapsulation in non-object oriented languages and in computer systems more generally.

A longer version of this position paper, including graphical figures of each scheme, will be available from `http://www.mcs.vuw.ac.nz/comp/Publications/` as `CS-TR-03-6`.

References

1. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA Proceedings*, November 2002.
2. Paulo Sérgio Almeida. Balloon Types: Controlling sharing of state in data types. In *ECOOP Proceedings*, June 1997.
3. Anindya Banerjee and Dave Naumann. Representation independence, confinement, and access control. In *Proceedings of ACM Principles of Programming Languages (POPL)*, pages 166–177, 2002.
4. Boris Bokowski and Jan Vitek. Confined types. In *OOPSLA Proceedings*, 1999.
5. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2003.
6. Dave Clarke and Tobias Wrigstrad. External uniqueness. In *Foundations of Object-Oriented Languages (FOOL)*, 2003.
7. Dave Clarke and Tobias Wrigstrad. External uniqueness is unique enough. In *ECOOP Proceedings*, 2003.
8. David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA Proceedings*, 1998.
9. Peter Grogono and Mark Gargul. A graph model for object oriented programming. *SIGPLAN Notices*, pages 21–28, July 1994.
10. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *OOPSLA Proceedings*, 2001.
11. C.A.R. Hoare and He Jifeng. A trace model of pointers and objects. In *ECOOP Proceedings*, 1999.
12. John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, November 1991.
13. Dan H. H. Ingalls. Design principles behind Smalltalk. *BYTE*, pages 286–298, August 1981.
14. P. Müller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. In *ECOOP Workshop on Formal Techniques for Java Programs*, number TR 269 in Technical Report. Fernuniversität Hagen, 2000.
15. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP Proceedings*, 1998.
16. Alan Cameron Wills. *Formal Methods applied to Object-Oriented Programming*. PhD thesis, University of Manchester, 1992.
17. Thomas Zimmermann and Andreas Zeller. Visualizing Memory Graphs. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001*, pages 191–204. Springer-Verlag, May 2001.