Flexible Exclusion Control for Composite Objects

Abdelsalam Shanneb

School of Computer Science and Engineering
The University of New South Wales
Sydney – Australia

shanneba@cse.unsw.edu.au

John Potter

School of Computer Science and Engineering
The University of New South Wales
Sydney – Australia

potter@cse.unsw.edu.au

Abstract

We present a simple approach for implementing flexible locking strategies in a system of components, which may themselves be composite objects. The approach is *flexible* in that a developer can defer the distribution of locks in the system until deployment: the choice of lock type and granularity may therefore depend on the operating environment. We only consider *exclusion control*; this includes mutexes, read-write locks and read-write sets, but does not cover state-dependent locking or transaction-based approaches. In general we express exclusion requirements as sets of conflict pairs on component interfaces; elsewhere (Potter, Shanneb and Yu 2004) we have demonstrated the effectiveness of a general-purpose exclusion lock that can provide any required exclusion. We presume knowledge of the dependency between the interface of a composite object and its internal components.

This work extends and simplifies the work on exclusion algebra for composite objects (Noble, Holmes and Potter 2000). Our major contribution is to distinguish between the control required internally and that provided externally. This clarifies the role of the so-called upward and downward mappings of the earlier work. We also offer a succinct mathematical basis for our model.

Keywords: Concurrency control, concurrent objects, composite objects, component-based systems, locking granularity

1 Introduction

As programmers, we are imbued with a mind-set attuned to a sequential model of program behaviour. On seeing a sequence of code statements, we naturally think of the effect of this code executing one step after another. Often the correctness of code depends on this sequentiality. When producing code that will operate in a multithreaded environment in which there are concurrent threads operating on a shared memory space, we need to prevent interference between concurrent threads potentially operating on the same data, that is, we need to guarantee *thread-safety* for our system.

In order to provide thread-safety for software components, the simplest approach is to force mutually exclusive access to the component interface. For example, in Java, we can declare the methods of a class to be synchronized, which has the effect of blocking calls on an instance of that class, whenever another thread has an active method call on the same object. The apartment model of COM also provides this ability to force a whole component to be singly threaded. However forcing single threadedness at a high-level may unnecessarily limit concurrent activity, which then restricts system responsiveness or efficiency in a multiprocessor environment.

To increase the potential for concurrent activity, we can adopt two approaches. First we can move the controls inside the components, so that they are closer to the critical sections of code where the sharing violations may occur. Second we can adopt a finer degree of control by enforcing pair-wise exclusion on conflicting method calls, such as with read-write controls. In this paper we only consider *exclusion control*; this includes mutexes, read-write locks and read-write sets, but does not cover state-dependent locking or transaction-based approaches. This second choice also presumes some knowledge of the internal implementation of the component: we need to know the conflicts between the different methods of the interface. We can of course adopt both of these approaches simultaneously, and provide finer grain locking internally rather than at the external interface.

The contribution in this paper is to provide a simple approach to reasoning about the degree of exclusion control required by a particular component when placed in a particular operating environment in which the potential for concurrent calls on the component is known. Components may provide their own locks at their interface. This reduces the internal concurrency potential for the component. We presume knowledge of the dependency relation between a composite object and its internal components. With this knowledge, and given a particular locking strategy for all the components, we show how to propagate internal exclusion requirements outward, and the potential for concurrency inward, thereby checking that all components have been provided with their required exclusion control.

This work derives from earlier work by Noble, Holmes and Potter (2000) on exclusion for composite objects.

Copyright © 2005, Australian Computer Society, Inc. This paper appeared at the 28th *Australasian Computer Science Conference (ACSC2005),* The University of Newcastle, Australia. Conferences in Research and Practice in Information Technology, Vol. 38. V. Estivill-Castro, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

That paper contributed a simple algebraic notation for describing exclusion requirements, that was termed the *algebra of exclusion*, and provided schemes for mapping these exclusion requirements downwards and upwards through an object-based language that permitted the expression of locks and composite object dependencies. Unfortunately the formulation of the relationships between layers in that paper appears somewhat confusing, and our contribution here is to simplify the model so that way the relations are, we hope, clearer. This is important to our overall goal of a simple declarative approach for exclusion control with composite objects. Furthermore the downwards composition approach of that paper relied on guessing the result of the mapping and then checking that the guess was correct; by working with concurrency potential rather than exclusion, we remove the guesswork in this paper. For a given component interface, we describe both exclusion and concurrency potential using the same notation used in the earlier paper, but now refrain from calling it the algebra of exclusion: in fact it is just a kind of graph algebra which allows the expression of undirected graphs, simultaneously describing both node and edge sets. We use this graph algebra for describing both exclusion requirements (the node set is the set of method names in the interface, and the edge set, the conflict pairs) and concurrency potential (pairs in the edge set denote allowable concurrency between pairs of methods in the interface).

In Section 2 we review related work. Section 3 introduces the way in which we talk about the exclusion requirements for the internals of a component to function correctly, any locking provided by that component, and the consequent external exclusion requirement, that must be provided by the component's operating environment (typically a composite object that it is part of). The relationship between the exclusion requirement and concurrency potential determines whether the component has sufficient control provided. Section 4 looks at how the component dependency relation determines how exclusion requirements are propagated outward, and how concurrency potential is propagated inward. Section 5 re-presents the major example of (Noble, Holmes and Potter 2000) to allow comparison of our new formulation with the earlier one. Section 6 presents a simple mathematical formalisation underpinning the previous sections. Section 7 critiques our approach, discusses limitations, and outlines our ideas for further work.

2 Related Work

Concurrency and synchronisation have always been attached to the object paradigm since its birth. Early languages and systems (Birtwistle, Dahl, Myhrhaug, and Nygaard 1979), (Hoare 1974) and (Brinch-Hansen 1973) had started adopting the object as the unit of synchronisation. Briot, Guerraoui and Lohr (1998) presented a comprehensive survey of systems that integrate concurrency and object-oriented languages.

Boyaparti and Rinard (2001) describe a type-system to enforce locking conventions in Java, based on the ownership type system of (Clarke, Potter and Noble 1998), (Clarke, Potter and Noble 2001), and (Clarke and Drossopoulou 2002). It is distinguished by enabling classes to be generic in their protection mechanisms, which are specified when instances are created. Protection is based on object ownership: every object has exactly one fixed owner that is specified through type parameterization. Before accessing a field of an object or invoking a method, the lock on the object at the root of the ownership hierarchy of the object must be held. Their use of ownership properties for restricting access and containment purposes is indirectly related to our approach in grouping of locks and in some cases restricting access through a single lock.

Greenhouse and Scherlis (2002) present their model for expressing design intent that may help programmers to enable assured consistency between design intent and code. Their "client policy' notation for describing safe/unsafe method interactions is analogous to our method-level exclusion specification for components of a composite.

Philippsen (2000) gives a comprehensive comparison survey of concurrent object-oriented languages in terms of identifying the key areas of integration as well as differences between the object-oriented and concurrent programming paradigms. In terms of performance comparisons between concurrent object-oriented systems based on locking granularity, we have found little evidence of published work, whereas much work investigating locking granularity is evident in the area of database systems as in (Rez 1995, Suh-Yin and Ruey-Long 1996). Elsewhere (Potter, Shanneb and Yu 2004) we have demonstrated the effectiveness of a general-purpose exclusion lock that can provide any required exclusion.

One of our research goals is to provide a heuristic model that puts some design decisions at the hands of programmers. These design issues are commonly hidden in the system, so we want to see these design decisions explicitly declared in component interfaces. Lea (1999) reveals the variety of approaches for designing and implementing concurrent programs in Java; it offers many techniques and patterns for letting threads work together safely. JSR-166 (2004), following on from the Java Concurrency Package of JDK 1.5, builds on the work of Doug Lea and others, and promises solutions to common special-purpose synchronisation problems. Our work is along this line in terms of distinguishing between the locking requirements of a design and the actual locking strategy implemented.

3 Control Layers

In this section we introduce our concepts of required exclusion and potential concurrency for a component, and also internal and external control. A component's interface is a set of its methods (Figure 1). The exclusion requirement for a component is specified as a set of method pairs that may conflict. Typically they depend on the internal dependencies of that component. The

component's exclusion requirements must be met to guarantee safe concurrent access to its interfaces. On the other hand, the potential concurrency for a component reflects its operating environment; it too is specified as a set of method pairs on the component's interface. So, our approach makes a clear difference between required exclusion and potential concurrency.

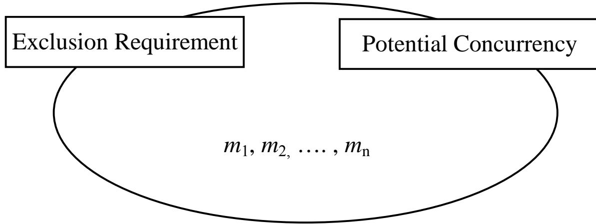

Figure 1: Required & Potential

Furthermore, we classify required exclusion and potential concurrency as external or internal. External required exclusion is that not provided by local locks and has to be supplied externally by the environment where the component is residing. Internal required exclusion, on the other hand, is determined by the required exclusion of a composite's components.

3.1 Exclusion Requirement

We depict the exclusion requirement as three distinct layers (Figure 2), an external layer R_E , an internal layer R_I and a provided local lock. The external requirement depends on the other two layers, summarizing whatever internal requirement is not provided by the local lock.

$$R_{External} = R_{Internal} - \text{ } Lock$$

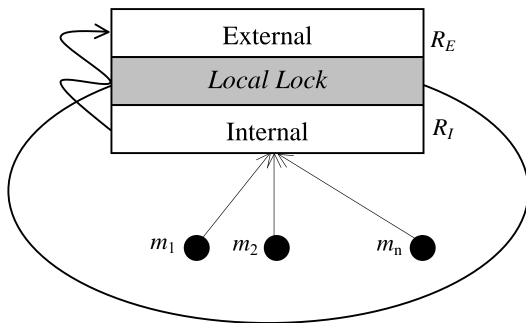

Figure 2: Required Exclusion

3.2 Potential Concurrency

The same idea applies to the composite's potential concurrency but in the opposite direction (Figure 3). All potential concurrency available internally (P_I) can be determined after extracting the local lock from the external potential concurrency (P_E) according to:

$$P_{Internal} = P_{External} - Lock$$

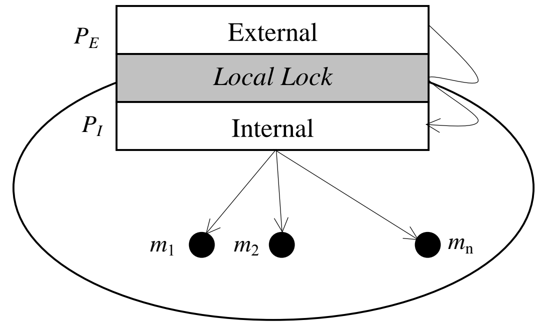

Figure 3: Potential Concurrency

3.3 An Example

An example may clarify the idea. We begin by presenting a notation for describing exclusion requirements. The expression (m_1 | m_2) describes two methods m_1 and m_2 that can execute in parallel. An interference between two methods is described as (m_1 x m_2), and an over bar on a method name ($\overline{m_1}$) indicates self-exclusion on that method, that is, only one thread may access that method. For example the expression ($\overline{m_1}$ | m_2) permits parallel invocations of m_1 and m_2 , but only one call m_1. Any exclusion expression may also be written as a conflict matrix:

	m_1	m_2
m_1	1	0
m_2	0	0

($\overline{\boldsymbol{m_1}}$ | $\boldsymbol{m_2}$)

Figure 4 depicts an example of exclusion requirements for three components. In C_1, the internal control is totally dependent on the external control since no local lock is provided. The set of layers in C_2 shows how the external environment needs to complement the local lock for providing the required internal control, and in C_3, no external control is needed as the local lock provides the internal requirement.

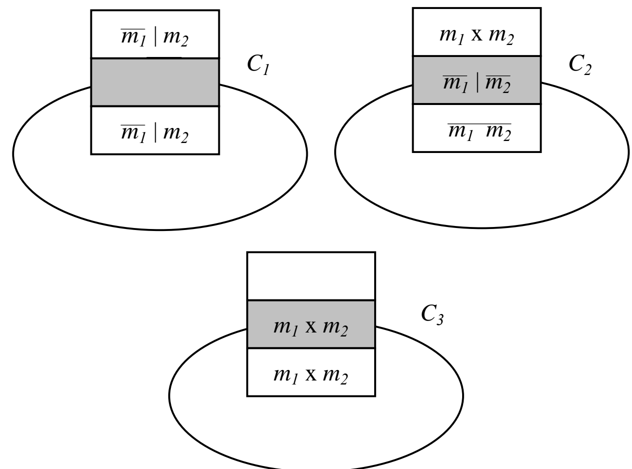

Figure 4: External Vs Internal

3.4 Hierarchical Model

The interface of a composite is the designated set of its methods. Figure 5 takes our component one step further by introducing composite objects with internal components. The figure also shows how we incorporate the required and provided requirements into the control layers.

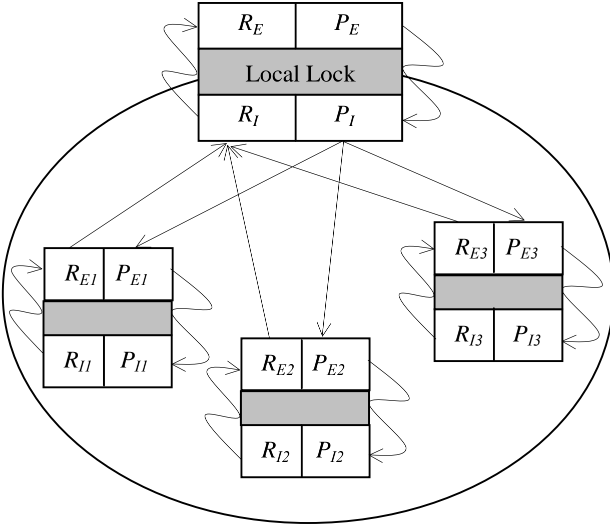

Figure 5: Hierarchical Composite

R = Required Exclusion P = Potential Concurrency

The previous figure brings our classification to completion as it shows how the internal exclusion requirement of a composite are constructed from the external exclusion requirements of the composite's internal components. Also, the figure shows how the external potential concurrency (if needed) is used for supplying the internal potential concurrency of the inner components. These relations between the required/potential and internal/external can be extended to any depth of the composite object. Also, we need to emphasize that a component is safe just when R_E and P_E (equivalently R_I and P_I) have no pairs in common.

4. Composition via Mapping

In this section we show how we use mapping functions to calculate exclusion requirements outwards from components to their container as well as the inwards from a container to its components.

4.1 Requirements Composition

In the previous section, we introduced the concept of internal exclusion requirements being satisfied by a combination of the exclusion requirements of the component's objects. We now explain the process of how we compose these internal exclusion requirements. Figure 6 starts an ongoing example depicting a composite with two interface methods I_1 and I_2, and two internal components C_1 and C_2 where each component has a couple of internal methods as shown. Let's also assume that each component is supplied with its exclusion requirements. We also assume that the usage pattern of the composite interface is given as follows; where *users(m)* is the set of interface methods that uses *m*.:

users(m_1) = {I_1}, users(m_2) = {I_2}, users(n_1) = {I_2} and users(n_1) = {I_1}

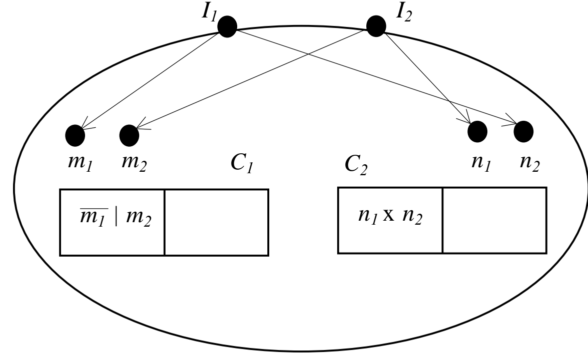

Figure 6: Component Dependency

We start by mapping the exclusion requirements of the internal components to the composite interface according to the composite usage pattern. This mapping process is simply achieved by substituting each method name in each inner component with the name of the composite interface that uses that method.

External requirement on inner component $c \Rightarrow$

Internal requirement on outer composite [*users(m)/m*] for each *m* in *c*.

For C_1: $\overline{m_1} \mid m_2$ $[I_1 / m_1, I_2 / m_2] \Rightarrow \overline{I_1} \mid I_2$

For C_2: n_1 x n_2 $[I_2 / n_1, I_1 / n_2] \Rightarrow I_2$ x I_1

We then combine the new mapped expressions:

$(\overline{I_1} \mid I_2) \mid (I_2 \text{ x } I_1) \Rightarrow \overline{I_1} \text{ x } I_2$

As a conflict matrix:

	I_1	I_2
I_1	1	0
I_2	0	0

|

	I_1	I_2
I_1	0	1
I_2	1	0

=

	I_1	I_2
I_1	1	1
I_2	1	0

So, the external exclusion requirements of the internal components collectively form a composition of requirements representing the internal exclusion requirements of the composite; $R_I = (R_{E1} \mid R_{E2}) = \overline{I_1}$ x I_2. Assuming that the local lock provides the needed internal required exclusion (R_I) (i.e $L = \overline{I_1}$ x I_2), and using the relation $R_{External} = R_{Internal} - Lock$, we find that the external exclusion requirement (R_E) is void in this case (Figure 7).

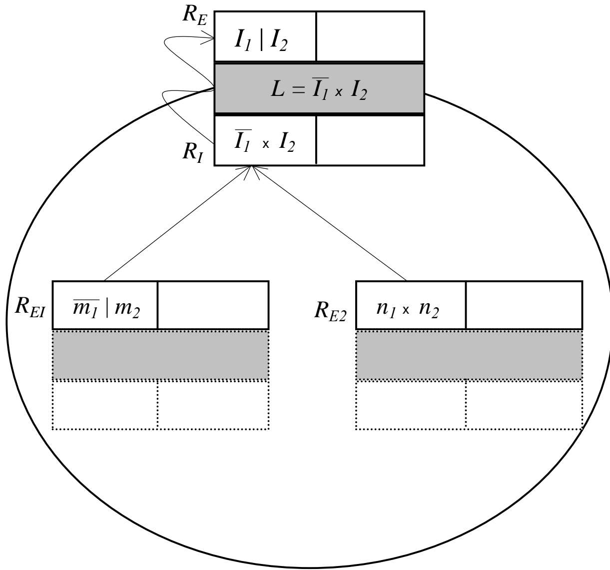

Figure 7: Requirements Composition

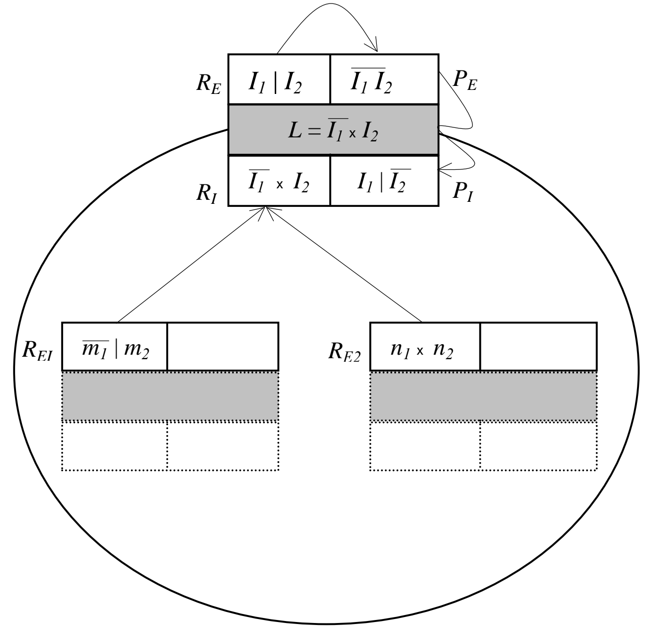

Figure 9: Internal Potential Concurrency

4.2 Potential Concurrency Calculation

The internal exclusion requirement of the composite can now be used to determine all potential concurrency allowable on this composite. External potential concurrency (P_E) is simply the complement of the external exclusion requirement (Figure 8).

Potential allowable concurrency $\Rightarrow (I_1 | I_2)^c = \overline{I_1 \times I_2}$

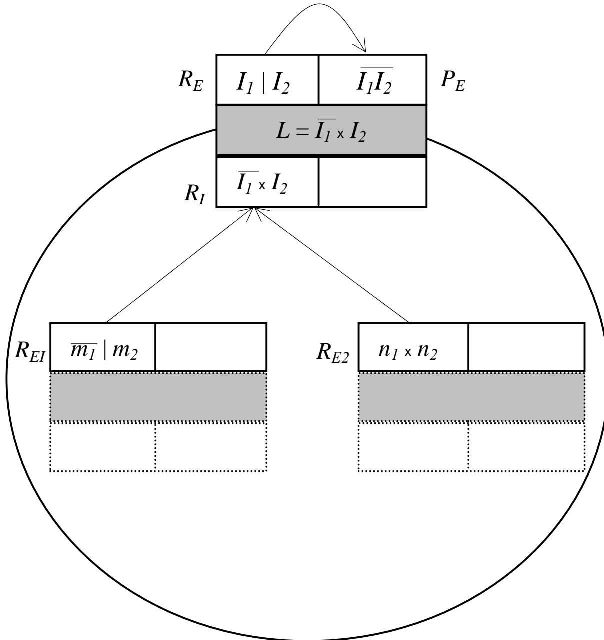

Figure 8: External Potential Concurrency

Having determined the composite's external potential concurrency, we use our previous formula (from 3.2) to calculate the internal potential concurrency for the composite main component (Figure 9):

$P_{Internal} = P_{External} - Lock \Rightarrow (\overline{I_1 I_2}) - (I_1 \times \overline{I_2}) = I_1 | \overline{I_2}$

Now we come to the last step of our process. Here we use the internal potential concurrency expression to obtain the potential concurrency for the inner components of the composite. We use a mapping function that makes use of the composite's interface usage patterns along with the composite's internal potential concurrency.

OuterAllowed[$used_by_c(I) / I$] , for each I of the composite interface, where *used_by* is the set of methods used by each I.

For $C_1 : I_1 | \overline{I_2} \Rightarrow m_1 | \overline{m_2}$

For $C_2 : I_1 | \overline{I_2} \Rightarrow n_2 | \overline{n_1}$ as shown in figure 10.

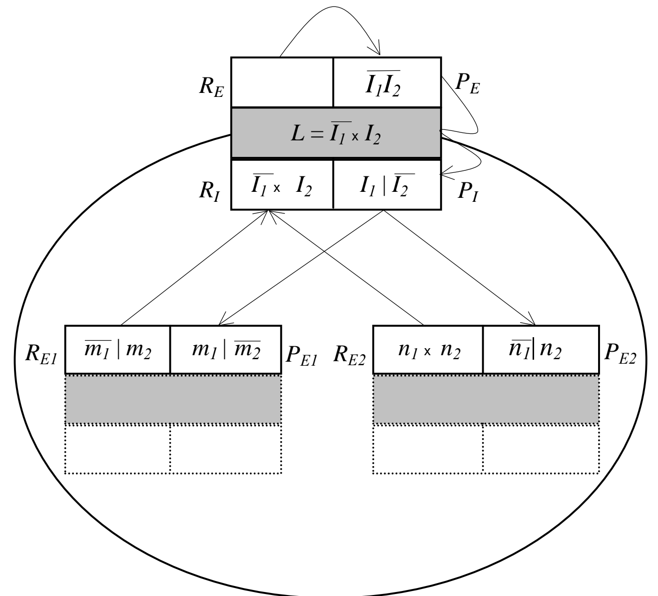

Figure 10: External Potential Concurrency for Components

4.3 Fine-Grain Locking

In this section we present another example to demonstrate our process. Here we apply our mapping techniques on a composite structure with more than two levels. First, we ought to emphasize the following: 1) we don't impose any concurrency restrictions on the outer environment, 2) main interfaces of the composite are entry points (method names) of the composite, in this case a_0 and a_1, and 3) we assume that inter-method uses relations are given. In this example we consider how our technique works when locks are only provided at the lowest level of a composite (Figure 11). The interface of the composite is provided by the method set a_0 and a_1, and the reset of the internal interfaces are given by the following uses relations:

Method	Uses
a_0	b_0 , c_1
a_1	b_1 , c_0
b_0	d_1 , e_0
b_1	d_0 , e_1
c_0	f_0 , g_0
c_1	f_1 , g_1

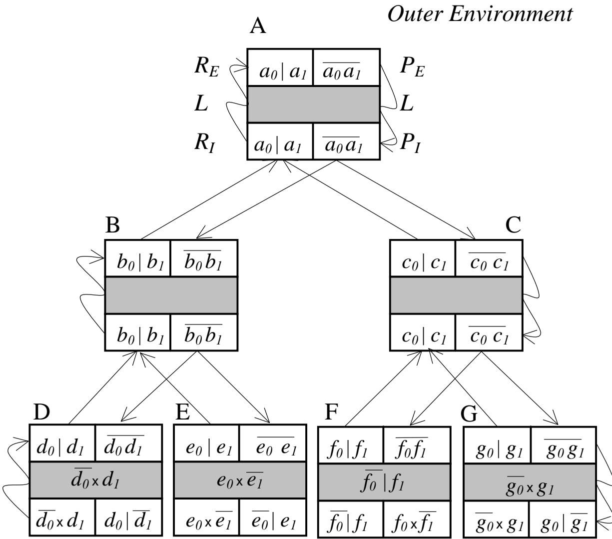

Figure 11: Locks at the bottom level

Figure 11 shows all the end relations according to the previous techniques. Remember that this example represents a fine-grain locking, that is, the internal components of this composite which represent the actual data objects are provided with local locks that meet their exclusion requirements. Our mapping technique starts by obtaining the exclusion requirements R_E of the external level of the internal components D, E, F, and G. We use the relation $R_{External} = R_{Internal} - Lock$ (from 3.1) to determine the external exclusion requirements. We find that no extra exclusion is needed at this level since the local lock provides the required internal exclusion; the external exclusion requirement for component D is written as $d_0 \mid d_1$.

We now use our mapping process (from 4.1) to determine the internal exclusion requirements for component B and C. Using the mapping we combine the external exclusion requirements of components D and E to calculate the internal exclusion requirements for component B, and combine the external exclusion requirements of F and G to calculate the internal exclusion requirements for component C. Again using the relation from 3.1, we then determine the external exclusion requirements for B and C. Composition for the internal exclusion requirement of component A is done in the same manner. After determining the external exclusion requirements of the top component which houses the composite interface to the outer environment, we found that $R_E = a_0 \mid a_1$, that is, no concurrency restriction is imposed on any access at that level, this is represented by the external potential concurrency $P_E = \overline{a_0 a_1}$ (from 4.2), which indicates every pair-wise interaction between a_0 and a_1 is allowed.

We invoke the relation $P_{Internal} = P_{External} - Lock$ to determine the internal potential concurrency of component A, and then use the mapping process from 4.2 to calculate the external potential concurrency for B and C.

After finishing all of the calculation and plugging in all the relations, we compare the internal exclusion requirement for each component at the bottom of our composite with its neighbor; the internal potential concurrency. That is, R_I's with P_I's. As previously mentioned, in order to guarantee component safety, all of the requirements R's and the potential concurrency P's either externally or internally should have no pairs in common. Looking at our resultant relations of the previous example, we find that this condition holds. At the same time comparing these pairs, we see that each R complements P in each box. We conclude that providing fine-grain locks ensures no excess exclusion.

4.4 Coarse-Grain Locking

The next example represents the other end of the spectrum where only one lock is imposed on our composite structure. Here, component A is provided with a lock which meets its internal exclusion requirements. Applying the same steps as in the previous example, we obtain the relations depicted in figure 12. Comparing the neighboring relations in the shaded box at the bottom of the figure, we find that we lose some concurrency potential. The allowable concurrency is $f_0 \times \overline{f_1}$ (the complement of the required exclusion), but the potential concurrency is only $f_0 \mid \overline{f_1}$. In other words the outer level control has unnecessarily excluded the concurrent activation of f_0 and f_1.

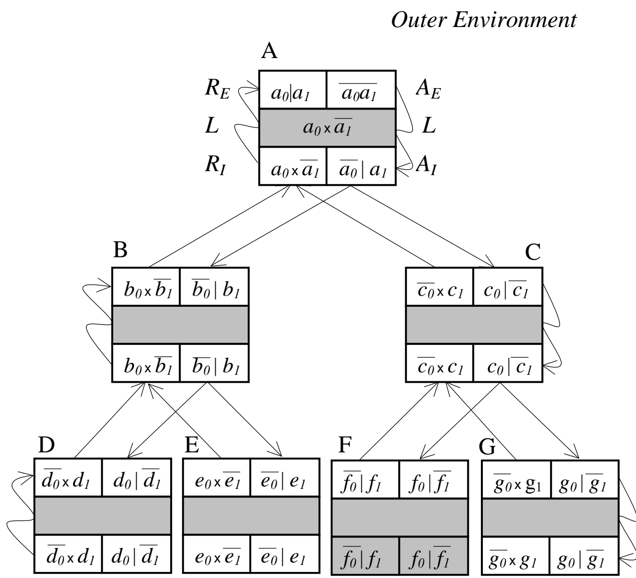

Figure 12. A lock at the top

5 A GUI Server Example

Figure 13 shows a composite system example from (Noble, Holmes and Potter 2000). This system illustrates the main components implementing a GUI server: a bitmap cache (also used to store font and icon information) that in turn uses RAM and disk cache subcomponents; an authentication component; an input queue that receives events from input devices; and an output queue that forwards rendering requests to graphics hardware. The queue objects are taken from a library (such as the Booch components (1990)) and can be parameterized with a strategy object to configure their locking behaviour. This graphics server is an encapsulated composition – the top GUI server object acts as a façade (Gamma, Helm, Johnson and Vlissides 1994) so that its internal component objects can't be accessed from outside, and each component either implements functionality internally, or invokes methods on their direct subcomponents.

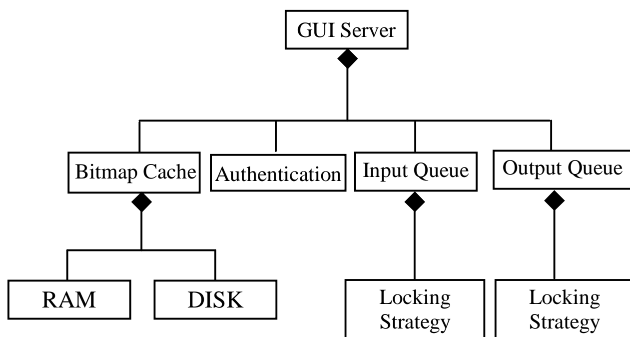

Figure 13. A GUI Composite

For this server to operate in a concurrent object-oriented environment, we must ensure that multiple threads accessing the server avoid interference, to protect the integrity of the components' data structures and invariants. There are a number of different approaches we can take:

- Ensure single threaded access by placing a single lock into the GUI server component.
- Allow maximally concurrent access to all components by placing locks on individual objects as necessary.
- Design an exclusion scheme for the whole server that uses individual locks to meet several components' requirements while maintaining a large amount of concurrency.

Lets apply our mapping techniques to determine the locking choices for this composite. We start by showing the usage pattern for internal cache components; namely, Disk and Ram:

Method	Uses
Cache.get (g)	*ram.get (g);* *disk.get (g)*
Cache.put (p)	*ram.put (p);* *disk.put (p)*

The usage pattern for the GUI server main interface is given as follows:

Server Methods	Uses
Server.login (l_i)	*auth.open (o)*
Server.logout (l_o)	*auth.close (c);* *inq.flush (f);* *outq.flush (f)*
Server.mouse (m)	*inq.enque (e)*
Server.draw (d)	*outq.deque (d)*
Server.cycle (c)	*inq.deque (d);* *auth.verify (v);* *cache.get (g);* *cache.pu t(p);* *outq.enque (e)*

After assuming that the uses relations are known, we also assume that exclusion requirments and some locks depicted in figure 14 are also given.

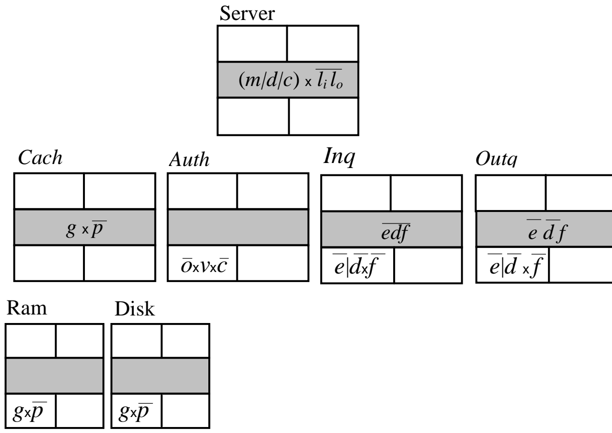

Figure 14: Given Requirements & Locks

The figure shows that not all components are provided with local locks or exclusion requirements. This layout illustrates some of the options for providing exclusion in this composition. For maximum concurrency, the leaf objects have their actual requirements given. We could apply precisely this amount of exclusion to them, ensuring safety but imposing runtime locking activity. To reduce this overhead, we can use information about the objects being designed to optimize their exclusion. The actual locks chosen for this example have been designed to seek a balance between execution overhead and granularity of exclusion.

After applying our outwards and inwards mappings as well as our internal-external relations, figure 15 shows all derived exclusions and potentials. The fact that there no exclusion requirement overlaps with any corresponding potential, shows that all components are safe. For the Server, the pairing of login with mouse and with draw is not required as part of R_I, but has been excluded by L, and do not appear as part of A_I. These are examples of lost concurrency potential. We see another example of lost potential between *enqueue* and *dequeue* for the Inq component.

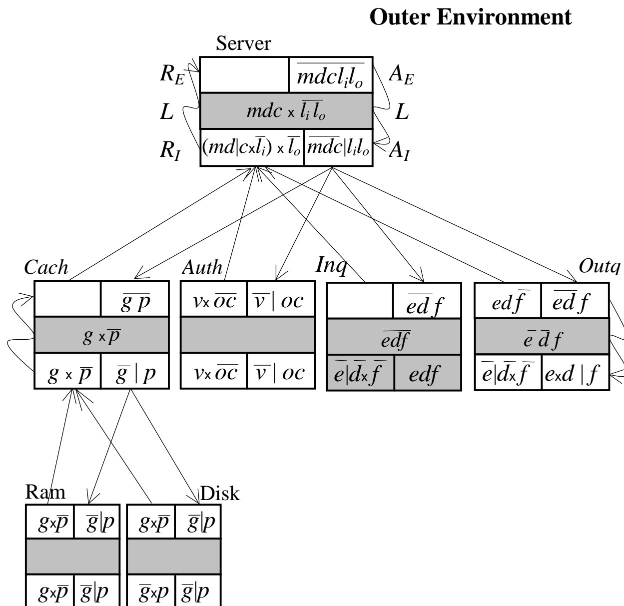

Figure 15: GUI Server

6 A Graph Algebra with Dependency Composition

In this section we formalise the notation and mappings presented in the previous sections. Exclusion requirements and concurrency potential have been expressed using a simple graph algebra in which the underlying domain is a set of names, interpreted as the names of methods. Here, for simplicity, we assume method names uniquely identify both the interface of a particular object, and the method in that interface. The dependency relation between composite objects and their components is presumed to be known.

The simple graph algebra used here is identical to the algebra of exclusion of objects (Noble, Holmes and Potter 2000) but we refrain from using that name here. We repeat the key definitions here for ease of reference.

e ::=	0	nothing
	n	name
	$e_1 \mid e_2$	disjunction
	$e_1 \times e_2$	product
	$\overline{e}$	completion

We interpret any such expression as a graph ($N(e)$, $E(e)$) where N is a set of names, and E is a set of unordered pairs of names. The following defines N and E for each construct of the algebra:

$$
\begin{aligned}
N(0) &= \{\} \\
N(n) &= \{n\} \\
N(e_1 \mid e_2) &= N(e_1) \cup N(e_2) \\
N(e_1 \times e_2) &= N(e_1) \cup N(e_2) \\
N(\overline{e}) &= N(e)
\end{aligned}
$$

and

$$
\begin{aligned}
E(0) &= \{\} \\
E(n) &= \{\} \\
E(e_1 \mid e_2) &= E(e_1) \cup E(e_2) \\
E(e_1 \times e_2) &= E(e_1) \cup E(e_2) \cup N(e_1) \otimes N(e_2) \\
E(\overline{e}) &= N(e) \otimes N(e)
\end{aligned}
$$

where $\otimes$ denotes symmetric Cartesian product.

Consider now a (directed) relation u defined on the underlying set of names. Given an undirected graph expression e we define its outward closure with respect to u as:

$$\uparrow_u e \;\hat{=}\; u^* \,.\, e \,.\, (u^{-1})^*$$

and the inward closure likewise:

$$\downarrow_u e \;\hat{=}\; (u^{-1})^* \,.\, e \,.\, u^*$$

Here the dot operator denotes forward composition of relations, and the star denotes reflexive, transitive closure.

We freely mix the notions of relation and the graph of a relation. The closure of *e* clearly contains *e*, closure is monotonic and idempotent.

We interpret *u* as the *uses* dependency relation between components. When *e* represents an exclusion requirement on a subset of the names, it is easy to see that its outward closure with respect to *u* represents all required exclusions induced on components *outside* the names of *e* (where a name *m* is *outside* another *n* if *m* uses *n* directly or indirectly). Because, if m_1 uses n_1, m_2 uses n_2 and n_1 is required to exclude n_2 then, in the absence of any provided exclusion inside, we must require m_1 to exclude m_2. Similarly when *e* represents concurrency potential, the downward closure maps the potential *inside* the names of *e*. Interestingly enough, these mappings apply even if there are cycles in the dependency relation.

For the kinds of system considered in earlier examples, the dependency relation is tree structured as far as components go, although any method of composite object may share any of its immediate components. Essentially this structure permits a layer by layer (modular) calculation of the closure operators. The dependency-based substitutions are simply an alternative way of presenting the calculation of the above closures, when the system, is layered (encapsulated) as in our examples.

In fact we can factor any dependency relation into such a structure by firstly identifying cycles, and factoring the relation over the cycles. If cyclic dependencies are present, all names that occur in a cycle need to be treated as equivalent as far as exclusion control goes. Cycles can therefore be factored out. We can then impose a hierarchical structure on the factored components by considering the dominator tree given the root names (the API for the system).

7 Critique and Conclusions

We have presented a simple approach for ensuring thread-safety for composite object systems in a flexible manner. Our model requires knowledge of the exclusion requirements on the interfaces of base-level components, the usage dependencies between the interface of each composite object and its components, and finally some expression of what the potential concurrent activation of the system might be in its operating environment. Then, given a particular distribution of locks throughout the components of the system, we can calculate whether or not each component is indeed thread-safe. This allows developers to design locking strategies separately from other implementation details, and allows flexibility in the distribution of locks that might be chosen. The actual choice of locks may be relative to the particular environment where the system (or component) is deployed. Furthermore, it is easy to determine where locks are redundant and where high level or coarse grain locks cause potential for concurrency to be lost.

To support our approach we have presented a simple mathematical model that justifies how the calculations are managed. This formalisation helps us to see how to deal with cycles and sharing in the dependency relation.

We note two limitations of our approach. First, we have not talked about state-dependent locks such as condition variables. We hope to pursue this in the future, but the key issue in dealing with such locks revolves around the nested monitor problem; such locks are inherently less flexible than exclusion-based locks. Second, we require knowledge of a composite's dependency on its components. This implies that we are talking about relatively static composite structures. However, with our work on ownership and related type systems, we are confident that we can use ownership type information to help reason about more dynamically structured systems. This too is a direction we intend to explore further.

One potential criticism of our approach is that it is based on the methods in an interface. Given that our model is phrased quite abstractly, we can choose to deal with any controllable program entity at all (e.g. particular critical sections of code, or even read or write access on individual program variables). We have merely presented our approach using methods and interfaces as a vehicle for the ideas. The key issue is that we must be able to reason about the uses dependency relationship for those entities that we wish to control.

8 References

Birtwistle, G., Dahl, O., Myhrhaug, B. and Nygaard, K. (1979): *Simula Begin*. Studentlitterature.

Booch, G. (1990): *Software Components with Ada*: *Structures,Tools,and Subsystems*.Benjamin/Cummings.

Brinch-Hansen, P. (1973): Operating Systems Principles. Prentice-Hall.

Briot, J., Guerraoui, R., and Lohr, K. (1998): *Concurrency and Distribution in Object-Oriented Programming. ACM Computing Surveys*, 30(3):291-329.

Boyapati, C. and Rinard, M. (2001): *A parameterized type system for race-free Java programs. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

Clarke, D., Potter, J., and Noble, J. (1998): *Ownership types for flexible alias protection. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

Clarke, D., Potter, J., and Noble, J. (2001): *Simple ownership types for object containment, In Proceedings of the 15th European Conference on Object-Oriented Programming* (ECOOP). 53 – 76.

Clarke, D., and Drossopoulou, S. (2002): *Ownership, encapsulation and the disjointness of type and effect,* In Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA). 292 – 310.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994): *Design Patterns*. Addison-Wesley.

Hoare, C. A. R.(1974): *Monitors: an operating system structuring concept.* Communications of the ACM, 17(10):549-557, Oct.

JSR-166: *Concurrency Utilities, Java Concurrency Process*. www.jcp.org. Accessed 1 Sept. 2004.

Lea, D. (1999): *Concurrent Programming in Java: Design Principles and Patterns*, (2nd Edition) Addison-Wesley.

Noble, J., Holmes, D. and Potter, J. (2000): *Exclusion for Composite Objects*, *Proceedings of ACM Conference on Object-Oriented Programming, Systems, and Languages. OOPSLA*, Minneapolis, USA.

Philippsen, M. (2002): *A Survey of Concurrent Object-Oriented Languages, Concurrency: Practice and Experience*. 12,917- 980.

Potter, J., Shanneb, A., and Yu, E., (2004): *Exclusion Control for Java and C#: Experimenting with Granularity of Locks*.CSJP'04 at the ACM PODC 2004.

Rez, Theo Harder. (1995): *Concurrency Control Issues in Nested Transactions with Enhanced Lock Modes for KBMSs*, DEXA'95, 6th International Conference and Workshop on Database and Expert Systems Applications.

Suh-Yin, Lee. and Ruey-Long, Liou. (1996): *A Multi-Granularity Locking Model for Concurrency Control in Object-Oriented Database Systems*. IEEE Trans. Knowl. Data Eng. 8(1): 144-156.