Exclusion requirements and potential concurrency for composite objects

Abdelsalam Shanneb\textsuperscript{a,}\textsuperscript{*}, John Potter\textsuperscript{a}, James Noble\textsuperscript{b}

\textsuperscript{a}Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
\textsuperscript{b}School of Mathematical and Computing Sciences, Victoria University of Wellington, Wellington, New Zealand

Received 1 November 2004; received in revised form 15 January 2005; accepted 1 March 2005
Available online 13 June 2005

Abstract

Concurrent object-oriented systems must prevent the interference that may arise when multiple threads simultaneously access shared components. We present a simple approach for implementing flexible locking strategies in an object-oriented system, in which the components themselves may be composite objects. We express exclusion requirements as sets of conflict pairs on component interfaces. Given knowledge of the dependency between the interface of a composite object and its internal components, we show how external exclusion requirements can be calculated from internal requirements, and further, how any potential concurrent activity outside an object can be projected into potential concurrency for the internal components.

With our approach we can defer the distribution of locks in the system until deployment: the placement of locks and choice of lock type for a component can depend on its operating environment. A Galois connection between the outward mapping of exclusion requirements, and the inward mapping of potential concurrency, limits how many locks are worth considering. In this paper we only deal with exclusion control, including mutexes, read–write locks and read–write sets, and do not cover state-dependent locking or transaction-based approaches.

© 2005 Elsevier B.V. All rights reserved.

\textsuperscript{*} Corresponding author.
E-mail addresses: shanneba@cse.unsw.edu.au (A. Shanneb), potter@cse.unsw.edu.au (J. Potter), kjx@mcs.vuw.ac.nz (J. Noble).

0167-6423/ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.03.004
1. Introduction

Whether by education, experience, or accident, most programmers habitually treat programs as sequential. When presented with a series of statements in almost any programming language, we are drawn to imagining the effect of executing these statements one step after another. Our assumptions about the correctness of code depend critically on this sequentiality. Unfortunately, our intuition does not model the execution of concurrent programs. In a multi-threaded environment, concurrent threads sharing resources can interfere with each other's execution.

To guarantee thread-safety for our systems, we need to prevent interference between concurrent threads potentially operating on the same data. In order to provide thread-safety for software components, the simplest approach is to force mutually exclusive access to the components' interface. For example, in Java, we can declare all the methods of a class as synchronised, serialising concurrent calls to each instance of that class so that each object acts as a monitor. COM’s apartment model similarly allows entire components to be singly threaded. Single-threading entire high-level components or subsystems necessarily limits concurrent execution, thereby restricting system responsiveness and efficiency in multiprocessor environments.

To increase the potential concurrency in a system while maintaining thread-safety, we can adopt two complementary approaches. First, we can move monitor boundaries from high-level components down to subcomponents, so that rather than single-threading an entire subsystem, only the shared objects within that subsystem are single-threaded. Second, we can adopt a finer granularity of exclusion control, such as read–write locks, rather than simply single-threading entire components. We can of course adopt both of these approaches simultaneously, and provide finer grain locking internally rather than at the external interface.

This article contributes a novel approach for reasoning about concurrency and exclusion in component-based object-oriented systems. We provide a simple notation for recording the exclusion requirements of each component in a system. Programmers can associate fine-grained exclusion policies with any object in the composition. Then, using dependency relations between composite and subsidiary components, we show how to propagate internal exclusion requirements outward, and the potential for concurrency inward, thereby checking that all components exclusion requirements have been met, ensuring that the system as a whole will be thread-safe. Our notation only addresses exclusion control; this includes mutexes, read–write locks and read–write sets, but does not cover state-dependent locking or transaction-based approaches.

This work extends our earlier work on the algebra of exclusion [26,28], by introducing an explicit notion of potential concurrency that is complementary to exclusion. The earlier approach was unable to calculate the exclusion that a composite component provided for its subcomponents: rather, it relied on programmers guessing the exclusion first, and
then verified the guess. By working with potential concurrency rather than exclusion, this article removes the guesswork. This is important to our overall goal of a simple declarative approach for exclusion control with composite objects. The recognition of a Galois connection between the outward mapping of exclusion requirements and inward mapping of potential concurrency is also new here.

This article is structured as follows. After briefly discussing the most salient work on concurrency in object-oriented systems in Section 2, we introduce our notation for exclusion and concurrency in Section 3. Section 4 shows how to propagate exclusion requirements outward, and potential concurrency inward. In Section 5, we describe how locks may be distributed within composite object structures, varying the potential for internal concurrency, while meeting exclusion requirements. The Galois connection is introduced in Section 6 where we demonstrate its use in identifying which locks (or parts thereof) are essential within a given distribution of locks.

2. Related work

Concurrency and synchronisation have always been attached to the object paradigm since its birth. Early languages and systems [3,16,6] had started adopting the object as the unit of synchronisation. We do not attempt a full survey here—see for instance Briot et al. [7] or Philippsen [27] for comprehensive surveys of systems and approaches that integrate concurrency and object-oriented languages.

Boyapati and Rinard [5] describe a type-system for enforcing locking conventions in Java, based on the ownership type system [11,10,9]. 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 parameterisation. 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. Jacobs et al. [19] are also making use of the ownership system in a verification technique for multithreaded object-oriented programs. 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 [15] present their model for expressing design intent that may help programmers to assure 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. Whereas their concern is to relate design intent to code, our focus is on calculating what locks are sufficient to satisfy given exclusion requirements in a given concurrent environment.

Two recent articles present the spectrum of modern, language-based approaches to synchronisation. Caromel et al. [8] describe a monitor based extension to Java that is similar to various other aspect-oriented synchronisation schemes [24,21,17,2]. Such schemes provide language extensions so that synchronisation or scheduling code can be executed whenever a method enters or leaves an object. Programming languages such as Polyphonic C♯ [1] and JoinJava [18] are at the other end of the spectrum—incorporating constructs from the Join Calculus [14] directly into programming languages.
In these languages, synchronisation policies are expressed using chords—combinations of synchronous and asynchronous methods whose execution implicitly establishes a rendezvous between multiple threads.

Both aspect-oriented and join-calculus languages can be used to implement a wide variety of concurrency management techniques, from basic exclusion, to state-dependent and transactional semantics. This control is provided, however, by writing code, rather than a declarative specification, so there is no notion of a separation of synchronisation policy and mechanism, and synchronisation policies can only be changed—say to distribute locks over composite objects—by changing code. We have found little evidence of published work dealing with the effect of locking granularity for concurrent object-oriented systems, whereas there is considerable related work evident in the area of database systems [29,23].

One of our research goals is to provide a declarative model for concurrency control that is easy to use, yet practical and efficient. Typically concurrency control is hard-coded within method bodies, which makes the control policy inflexible, and contributes to the so-called inheritance anomaly [25]; our declarative and wrapper-based implementation approach avoids placing synchronisation code inside classes. Lea [22] reveals the variety of approaches for designing and implementing concurrent programs in Java. Lea and others have developed new concurrency utilities for Java [20], illustrating the practical importance of flexible concurrency control mechanisms. Elsewhere [28] we have demonstrated the effectiveness of a general-purpose exclusion lock that can provide any required exclusion.

3. A notation for expressing exclusion and concurrency

In this section we introduce a notation for expressing exclusion requirements on a component. The same notation can express the potential for concurrent activity in the component’s environment. Consider the example of an object with three interface methods \(m_1\), \(m_2\), and \(m_3\) and two internal fields \(v_1\) and \(v_2\). Let us assume that these fields are not atomic, so they need protection from concurrent access. Method \(m_1\) writes to the field \(v_1\), \(m_2\) writes to \(v_2\), and \(m_3\) reads from both \(v_1\) and \(v_2\); so, \(m_1\) and \(m_2\) are independent writers, and \(m_3\) is a reader interfering with both, as shown in Fig. 1(a). The matrix in Fig. 1(b) shows the internal conflicts for read and write access on the two fields; the non-interference or independence of \(r_1\), \(w_1\), and \(r_2\), \(w_2\) is evident from the zero entries in the off-diagonal blocks.

Rather than use conflict matrices to express exclusion requirements, we choose to adopt a more succinct algebraic notation that we call the algebra of exclusion [26].
We will elaborate on this notation shortly. By way of illustration, we write the exclusion requirements for the two fields as: \( r_1 \times \overline{w_1} \mid r_2 \times \overline{w_2} \). This exclusion requirement on the internal fields can be reflected at the level of the component interface: because \( m_1 \) and \( m_2 \) each write to different fields, any calls should execute in self-exclusion (that is, separate calls on the same method should be blocked to avoid concurrent write access to the same field), but there is no need for them to execute in mutual exclusion. However \( m_3 \) is only a reader, so does not require self-exclusion, but because \( m_3 \) reads both fields, its activity conflicts with that of \( m_1 \) and \( m_2 \). This is shown in the conflict matrix of Fig. 1(c), with the corresponding exclusion expression: \( m_3 \times \overline{w_1} \mid m_3 \times \overline{w_2} \). In Section 4 we will see that this expression is calculated from \( r_1 \times \overline{w_1} \mid r_2 \times \overline{w_2} \), via the substitution \([m_3/r_1, m_1/w_1, m_3/r_2, m_2/w_2]\) determined from the columns of the usage relation of Fig. 1(a).

The primitive elements of the algebra of exclusion are (method) names, and an expression identifies both an underlying set of names (the vertices of a graph) and a symmetric binary relation on that set (the undirected edges of the graph). So the algebra of exclusion is just a notation for defining finite undirected graphs. There are two main operators. A disjunction \((a \mid b)\) combines the names of \( a \) and \( b \) without introducing any (extra) pairs; this is the same as graph union. A product \((a \times b)\) combines its names and introduces the (unordered) pair \((a, b)\). When interpreted as exclusion requirements, \( a \times b \) means that \( a \) and \( b \) conflict (that is, form an exclusion pair). The expressiveness of the notation comes from allowing these operators to be used on arbitrary expressions, and not just between primitive names—their precise meaning is given in Appendix A. We use \( \overline{a} \) to indicate the self-exclusion \( a \times a \). Not only do exclusion expressions denote a set of exclusion pairs, but they also define all names in the interface; for example, both \( \overline{a} \) and \( \overline{a} \mid b \) denote the self-exclusion on \( a \), but the latter is for an extended interface that includes \( b \) independently of \( a \).

For syntactic convenience we assume that product (\( \times \)) takes precedence over disjunction (\( \mid \)), so \( a \mid b \times c \) is equivalent to \( a \mid (b \times c) \). We also use concatenation of expressions as a form of disjunction with higher precedence, so the expression \( ab \times c \) is equivalent to \( (a \mid b) \times c \).

The exclusion expression \( abc \) denotes a mutex on its three methods in which all pairs are conflicting; it is equivalent to \( a \mid b \mid c \) and also to \( a \times b \times c \). We can think of this as the strongest exclusion requirement, or equivalently the coarsest grain lock that we can provide. Less restrictively, the expression \( a \times \overline{bc} \) represents a read–write lock with \( a \) as a reader and \( b \) and \( c \) as writers. Even less restrictively we may have \( c \) as an independent writer that can execute concurrently with either of the others. We can express this as \( a \times \overline{bc} \mid c \); this exclusion represents the independent combination of a read–write lock on a reader \( a \) with writer \( b \), and an independent mutex on \( c \). The finest grain control is vacuous, allowing all methods to execute concurrently; we denote this simply as \( abc \) (or equivalently, as \( a \mid b \mid c \)).

Despite its name, we also use the algebra of exclusion for writing expressions describing concurrent method calls. In fact, given an exclusion requirement \( e \), its complement, \( e^c \), denotes exactly those pairs of methods which can safely be activated concurrently—that is, all pairs of methods which do not exclude each other. For example, the read–write lock given by the exclusion expression \( a \times \overline{bc} \) has complement \( \overline{a} \mid bc \); \( a \) is the only method which may be executed concurrently. The more complex lock given by \( a \times \overline{bc} \mid \overline{c} \) has complement...
\(ab \times c\): single invocations of \(c\) may execute concurrently either with multiple invocations of \(a\) or with single invocations of \(b\).

Where necessary we use the terms exclusion expression and concurrency expression to disambiguate the intended interpretation.

4. Exclusion requirements and potential concurrency

In this section we relate the concept of potential concurrency \((P)\) and exclusion requirements \((R)\) for a component. We consider the interface of a component as a set of method names. As we just saw in Section 3, the exclusion requirement for a component is specified as a set of method pairs that may conflict. Typically they depend on the internal implementation of that component, and in particular on any sharing conflicts that occur internally. The exclusion requirements of the component 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, representing those methods which can have concurrently active calls in the environment—that is, those methods which are potentially concurrent.

So, in our approach we make a clear difference between exclusion requirements and potential concurrency—see Fig. 2(a). Exclusion requirements are determined by the internal implementation of a component; potential concurrency is determined externally to a component. We illustrate these ideas with our example from Fig. 1(c). Here, the exclusion requirement, \(R = m_3 \times |m_1| m_3 \times |m_2|\), is simplified to the equivalent expression \(m_3 \times (m_1 | m_2)\) as shown in the left-hand box of Fig. 2(b). The expression \(P = m_1 m_2 m_3\) in the right-hand box represents the potential concurrency available in the outer environment; in this example \(P\) represents an unrestricted environment with maximal potential concurrency. This example illustrates an unsafe situation in which none of the exclusion requirements of the component have been provided, and there are potentially unsafe calls in the environment.

In a single-threaded environment, in which there is no potential concurrency, we would write \(P = m_1 m_2 m_3\). Even though no locks are provided, this system is safe. In general terms, we say that a system is safe if its exclusion requirement and potential concurrency have no pairs of methods in common.

4.1. Exclusion requirements composition

The exclusion requirements of a composite object can be determined by composing the exclusion requirements of its internal components, as we explain next. Fig. 3 depicts a composite with two interface methods \(k_1\) and \(k_2\), and two internal components \(C_1\) and \(C_2\), where each component has a couple of methods and known exclusion requirements as shown. We assume the usage relation: \(k_1\) uses \(m_1\), \(k_1\) uses \(n_2\), \(k_2\) uses \(m_2\) and \(k_2\) uses \(n_1\). To map the inner layer exclusion requirements to the outer layer, we replace each occurrence of an inner method name with the set of outer methods that use that inner method:

- Inner exclusion [outer/inner] \(\Rightarrow\) Outer exclusion
  
  \[m_1 \mid m_2 \{k_1/m_1, k_2/m_2\} \Rightarrow k_1 \mid k_2\]
  \[n_1 \times n_2 \{k_2/n_1, k_1/n_2\} \Rightarrow k_2 \times k_1\]
Why does this work? Consider component $C_2$. Because $k_2$ uses $n_1$ and $k_1$ uses $n_2$, if $k_2$ and $k_1$ are activated concurrently, it is possible for $n_1$ and $n_2$ to conflict. Putting this around the other way, the exclusion requirement on $n_1$ and $n_2$ induces an exclusion requirement on $k_2$ and $k_1$. In general, every pair of inner methods that must be excluded induces an exclusion requirement on their respective users. The two expressions, $k_1 | k_2$ and $k_2 \times k_1$, represent the exclusion requirement induced on the composite object by each of its components. Because the components are independent of one another, their composition produces no further conflict.

In terms of conflict matrices, this simply combines the conflicts caused by each of the inner components (Fig. 4). So the outer exclusion requirement is: $R = (k_1 | k_2) | (k_2 \times k_1) = k_1 \times k_2$.

In summary, as in Fig. 5, the schematic form for mapping of composite inner exclusion requirement $R_1 | R_2$ to an outer layer exclusion requirement $R$ is

$$R = (R_1 | R_2) [outer/inner]$$
This form relies on the two components being independent; if the two components share access to other components, then we need to use their joint exclusion requirement, derived from their shared usage, instead of the modular form $R_1 \mid R_2$.

4.2. Potential concurrency decomposition

If a composite object is deployed in an environment with known potential for concurrent activations of its methods, then, given the composite structure, we can determine the potential concurrency for the inner components from the outer potential concurrency. We illustrate it with the composite object of Fig. 3.

Suppose the potential concurrency is given as $P = k_1 \mid k_2$; in other words, we have an environment making calls on $k_1$ and $k_2$, but restricted so that only concurrent activations are possible for $k_2$. In fact, this is the complement of the composite exclusion requirement $R = k_1 \times k_2$ that we just calculated, so the object is safe in this environment. Let us focus on the concurrency potential $P_1$ for inner component $C_1$. Since $k_2$ only uses $m_2$ on $C_1$, the only concurrency possible for $C_1$ is with $m_2$. In fact, for each inner component, we can calculate the potential concurrency directly by substitution—see Fig. 6:

$$\text{outer concurrency [inner/outer]} \Rightarrow \text{inner concurrency}$$

$$k_1 \mid k_2 [m_1/k_1, m_2/k_2] \Rightarrow m_1 \mid \overline{m_2}$$

$$k_1 \mid k_2 [n_2/k_1, n_1/k_2] \Rightarrow \overline{n_1} \mid n_2$$
We can also calculate a combined concurrency expression for the inner components: 
\[ k_1 \mid k_2 [m_1 n_2 / k_1, m_2 n_1 / k_2] = m_1 n_2 \mid m_2 n_1. \]
This is not simply \( P_1 \mid P_2 \) because \( m_2 \) and \( n_1 \) may run concurrently on the independent components. In fact each \( P_i \) is the restriction of this combined internal concurrency expression to the \( C_i \) interface. However, in general, we are less interested in the potential concurrency between independent components, and therefore have no need to consider the combined concurrency expression.

In summary, the schematic form for the mapping of the outer potential concurrency \( P \) to inner potentials \( P_i \) is
\[ P_i = P \left[ \text{outer} / \text{inner} \right]. \]

5. Distribution of locks

Exclusion requirements can be met by providing locks that restrict access to components. Locks may be of different types, such as mutex or read–write controls; we use exclusion expressions to specify the type of lock provided for a component. We can attempt to increase potential concurrency within an object by choosing finer grain locks. For composite objects, an alternative is to place locks on the inner components—by moving locks inwards, we avoid losing too much potential concurrency within the composite. In this section we complete our model by providing locks on each component of a system.

We now incorporate three exclusion expressions for components in our model (see Fig. 7): an internal exclusion requirement \( R_I \), derived from the internal structure of a composite, using the techniques of Section 4; a local lock \( L \), specifying the exclusion provided at the interface to the component; and an external exclusion requirement \( R_E \), capturing that part of the requirements not provided by the local lock. \( R_E \) can be calculated from the other parts: \( R_E = R_I - L \), summarising the missing exclusion for the component. Safety for a component with a non-trivial external exclusion requirement can only be achieved by ensuring that its external environment is suitably restricted. This separation between the internal requirement, the locally provided lock, and the missing exclusion to be provided externally is the key to our approach for reasoning about exclusion locking strategies.

Conversely, see Fig. 8, the potential concurrent activity, \( P_E \), outside of a component, will be restricted by the blocking behaviour of any local lock \( L \). The reduced potential concurrency internally \( P_I \) can be calculated as \( P_I = P_E - L \).
Fig. 8. External versus internal potential concurrency.

Fig. 9. External exclusion with varying local locks.

Fig. 9 illustrates the effect of different locks on the external exclusion requirements. In all examples, the internal exclusion requirement is the same. In C₁, the internal exclusion requirement is reflected externally, since no local lock is provided. In C₂ the external requirement is reduced by two individual mutexes, and in C₃, no external control is needed as the local read–write lock completely provides the required exclusion. In Fig. 10 we present the whole hierarchical picture for a composite object. The innermost exclusion requirements must be given. For all components, the internal exclusion requirement \( R_I \) may be partially met by the locally provided lock \( L \), with any missing exclusion requirement \( R_E = R_I - L \) propagated as the internal exclusion requirement of the next layer out, via the component usage relation. This model allows us to keep track of the effect of different locks and lock placement. The dual of this applies to potential concurrency. We presume the potential concurrency is specified at the outermost level (the operating environment for the system), which is then restricted by local locks and component usage as we move inwards through the components of the system. Clearly this approach works through any number of layers of a hierarchical composite system.

With this breakdown of exclusion requirements and potential concurrency, we can identify further interesting properties. We have already seen that the expression \( R_I - L \) denotes any missing exclusion requirements not met by the locally provided lock. \( L - R_I \) on the other hand, represents unnecessary blocking imposed by the lock on the component. The complement of the exclusion requirement, \( R_E^c \), is the maximum allowed concurrency for a component outside of the lock. Similarly \( R_I^c \) is the maximum allowed concurrency internally. We reiterate our safety criterion: a component is safe just when \( R_E \) and \( P_E \) have no pairs in common. Equivalent formulations of this are: \( R_I \) and \( P_I \) have no pairs in common; \( P_E \subseteq R_E^c \); and \( P_I \subseteq R_I^c \).
5.1. Different locking levels

We illustrate the calculations for mapping exclusion requirements outwards and potential concurrency inwards, with different types of locks at different levels. In Fig. 11 fine-grain locking is provided at the innermost level, which should allow the maximum potential for concurrency. The figure shows the result of calculating the exclusion requirements and potential concurrency for all components, assuming:
Methods | Uses | Used By
--- | --- | ---
$a_0$ | $b_0, c_0, c_1$ | 
$a_1$ | $b_1, c_0$ | 
$a_2$ | $b_0, c_0, c_2$ | 
$b_0$ | $d_0, e_0$ | $a_0, a_2$ | 
$b_1$ | $d_0, a_1, e_1$ | $a_1$ | 
$c_0$ | $f_0, g_0$ | $a_0, a_1, a_2$ | 
$c_1$ | $f_1, g_0, g_1$ | $a_0$ | 
$c_2$ | $g_0, g_1$ | $a_2$ | 
$d_0$ | | $b_0, b_1$ | 
$d_1$ | | $b_1$ | 
$e_0$ | | $b_0$ | 
$e_1$ | | $b_1$ | 
$f_0$ | | $c_0$ | 
$f_1$ | | $c_1$ | 
$g_0$ | | $c_0, c_1, c_2$ | 
$g_1$ | | $c_1, c_2$ |

Fig. 12. Component usage.

- the innermost exclusion requirements are given;
- maximum potential concurrency at the outermost environment is assumed;
- the only external entry points are $a_0, a_1$ and $a_2$ of the outermost interface;
- the only locks provided are those shown;
- the component usage is known (see table in Fig. 12).

Our exclusion calculations start with the requirements $R_E$ of the external level of the innermost components $D, E, F,$ and $G$, using $R_E = R_I - L$. For all these components, the local lock provides the required exclusion, so no further exclusion is required in the middle layer at $B$ or $C$, or the outer layer, at $A$. This is confirmed by the propagation of these exclusion requirements via the component usage relation. For example, for $B$, we find

$$R_{IB} = d_0d_1 [b_0b_1/d_0, b_1/d_1] | e_0e_1 [b_0/e_0, b_1/e_1]$$

$$= b_0b_1 | b_0b_1$$

$$= b_0b_1.$$  

At the outermost level $R_E = a_0a_1a_2$; that is, the exclusion requirement is vacuous here. The maximum allowed concurrency is $R_E^* = a_0a_1a_2$ which is identical to the given potential concurrency $P_E$. Thus the system is indeed safe, as expected. Conversely we map the potential concurrency inwards. Maximum potential concurrency is retained as far as the external level of the innermost components $D, E, F,$ and $G$, because there are no locks, and all methods are used.

At the internal level of the innermost components, the potential concurrency is restricted by the local locks provided there. Comparing the exclusion requirement with the internal
and external potential concurrency for all components, we see that they are always complementary. Hence we see that fine-grain locks guarantee no loss of potential concurrency.

The next example represents the other end of the spectrum where only one lock is imposed on the composite. Here, component $A$ is provided with a lock which meets its internal exclusion requirements, so the system is safe. Fig. 13 shows the result of all exclusion and concurrency calculations. Observe that at all levels, there is no overlap between the exclusion requirement and potential concurrency—all levels satisfy the safety criterion. For this example components $C$ and $E$ display lost potential for concurrency: $c_1$ is allowed to execute concurrently with $c_2$, as is $e_0$ with $e_1$, but cannot.

5.2. A GUI server example

Consider a more practical example, a graphical user interface (GUI) server, taken from our earlier work [26]. Fig. 14 shows the main components of the 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 [4]) and can be parameterised 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 [13] so that its internal component objects cannot be accessed from outside, and each component either implements functionality internally, or invokes methods on their direct subcomponents.

For this server to operate in a concurrent 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:
- enforce single-threading with a single mutex on the GUI server component;
- allow maximally concurrent access to all components by placing locks on individual components 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.

Fig. 15 shows usage of the GUI server components, and the cache usage of the disk and RAM components is shown in Fig. 16.

In this example we will validate the safety of a given locking policy against specified innermost exclusion requirements. These are given in Fig. 17. Not all components are provided with local locks or explicitly specified exclusion requirements. This figure suggests some of the options for providing exclusion in this system. For maximum concurrency, the innermost components could have locks providing precisely their required exclusion, ensuring safety but imposing runtime lock acquisition overhead. To reduce this overhead, we can use information about the objects being designed to optimise their

---

Table: GUI Component Usage

<table>
<thead>
<tr>
<th>Method</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>server.login(l_l)</code></td>
<td><code>auth.open(o)</code></td>
</tr>
<tr>
<td><code>server.logout(l_o)</code></td>
<td><code>auth.close(c); inq.flush(f); outq.flush(f)</code></td>
</tr>
<tr>
<td><code>server.mouse(m)</code></td>
<td><code>inq.enque(e)</code></td>
</tr>
<tr>
<td><code>server.draw(d)</code></td>
<td><code>outq.deque(d)</code></td>
</tr>
<tr>
<td><code>server.cycle(c)</code></td>
<td><code>cache.get(g); cache.put(p); auth.verify(v); inq.deque(d); outq.enque(e)</code></td>
</tr>
</tbody>
</table>

Table: Disk and RAM Usage

<table>
<thead>
<tr>
<th>Method</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cache.get(g)</code></td>
<td><code>ram.get(g); disk.get(g)</code></td>
</tr>
<tr>
<td><code>cache.put(p)</code></td>
<td><code>ram.put(p); disk.put(p)</code></td>
</tr>
</tbody>
</table>

---
exclusion. The actual locks chosen for this example have been designed to achieve a balance between execution overhead and granularity of exclusion.

Fig. 18 shows all derived exclusion requirements and potential concurrency. Because no exclusion requirement overlaps with any associated potential concurrency, all components are safe. This example also illustrates some lost potential for concurrency. For the server, the combination of login with both mouse and draw is allowable internally, but is locked out. Another example of lost potential occurs between enqueue and dequeue for the Inq component.

6. Inner–outer safety defines a Galois connection

The mapping of inner exclusion requirements outwards, and outer potential concurrency inwards, given the component usage relation for a composite object, exhibits a deeper mathematical property—a Galois connection. The properties of Galois connections allow us to classify the exclusion requirements and potential concurrency in an orderly way. In particular we can use this information to restrict how many locks we need to consider; essentially we are able to group locks into equivalence classes, and for each class restrict attention to minimal representatives.

An inner exclusion requirement \( R = R_{E_{\text{inner}}} \) on the components of a composite object already takes locks of the internal components into account. It can be propagated to an internal exclusion requirement \( R_{1_{\text{outer}}} \) for the object, via the inner–outer usage relation. The complement of this is the object’s internal allowed concurrency \( A_{1_{\text{outer}}} \). We use \( \prec \) to denote this mapping, combining inner–outer usage, and complement. So \( A_{1_{\text{outer}}} = R_{1_{\text{outer}}} = R \prec \). For safety at the outer level, the concurrency potential \( P = P_{1_{\text{outer}}} \) must not exceed the maximum allowable, so we require \( P \sqsubseteq R \prec \), as discussed in Section 5.

Furthermore we can propagate \( P \) inwards to calculate the concurrency potential for the internal components, the complement of which is the collection of excluded activation pairs, \( P_{E_{\text{inner}}} = P^\prec \). For safety at the inner level, the excluded concurrent activations
for the components must contain all the required exclusion: \( R \subseteq P^{\succ} \). It is not hard to demonstrate the inner–outer correspondence:

**Inner Safety holds if and only if Outer Safety holds**

which is formalised as

\[
R \subseteq P^{\succ} \iff P \subseteq R^{\prec}.
\]

This is precisely the statement that the pair of mappings \((\succ, \prec)\) is a Galois connection \([12]\). It follows that \(\prec\) and \(\succ\) are closure operators for inner exclusion requirements and outer concurrency potential respectively.

Furthermore, there is an order isomorphism between the fixed points of these closure operators, \(\{R^{\prec}\}\) and \(\{P^{\succ}\}\), where \(R\) varies over all possible inner exclusion requirements and \(P\) over all outer concurrency potentials. Because \((\succ, \prec)\) is a Galois connection \([12]\), given an inner exclusion requirement \(R\), its corresponding fixed point \(R^{\prec}\) is the maximum exclusion expression that maps to the same outer allowed concurrency as does \(R\), that is \(R^{\prec} = R^{\prec}\). By finding these fixed points, we partition the inner exclusion expressions into equivalence classes, each with a unique maximal representative. Similarly we can partition the concurrency potentials of the outer layer. Furthermore the two partitions of exclusion and concurrency potential expressions are order isomorphic.

The table in Fig. 19 shows the fixed point expressions for the Galois connection between composite B and its components D and E for the example of Fig. 11. In this example, every possible form of outer concurrency expression appears as a fixed point expression, which implies that the internal behaviour is sensitive to every change in the external potential.
concurrency. However few of the possible forms of inner exclusion expression appear, which we can exploit to reduce the number of locks we need to consider. Suppose we have, as in Fig. 11, an exclusion requirement \( d_0 \times d_1 \) on component D, and \( e_0 \times e_1 \) on E. Then the maximum allowed concurrency for B is \( (d_0 \times d_1) | e_0 \times e_1 \). Although it is simple enough to calculate the result, Fig. 19 gives us the alternative of table lookup. We find the least fixed point which covers the given expression, \( d_0 \times d_1 | e_0 \times e_1 \) with corresponding outer maximal allowed concurrency \( b_0 \times b_1 \), complementing the outer read–write exclusion requirement \( b_0 \times b_1 \) of Fig. 13.

But what if we provide internal locks? By providing the lock \( d_0 \times d_1 | e_0 \times e_1 \), we fully meet the inner requirement—the missing exclusion is given by \( d_0 \times d_1 | e_0 \times e_1 \), as in Fig. 11; in other words, there is no external requirement for this component. More interestingly, we can determine the minimal amount of locking \( L \) that will cause some change in the outer requirements by seeing when the missing inner requirement \( R_E = R_I - L \) is covered by a smaller fixed point in Fig. 19. For this example, \( L = d_0 \times d_1 \) reduces \( R_E \) to \( d_0 \times d_1 | e_0 \times e_1 \), which is covered by the inner fixed point \( d_0 \times d_1 | e_0 \times e_1 \) with the corresponding outer fixed point \( b_0 \times b_1 \). So with a simple pairwise lock on \( d_0 \) and \( d_1 \) we have removed some of the outer exclusion requirement (which is the complement of the outer fixed point). Without the internal lock, the outer requirement is the read–write expression \( b_0 \times b_1 \); with the internal lock it reduces to the single method mutex \( b_0 \times b_1 \). This will not be reduced by any further inner locking, until the full requirement is met.

In summary, the Galois connection between inner exclusion and outer concurrency permits us to precisely identify those increases in inner locks which will reduce the outer level exclusion requirements. This approach applies through more than one level of composition. In practice, we only need to consider that part of the Galois connection relevant for the given exclusion requirement and/or concurrency potential: given an inner exclusion requirement, we only need that part of the Galois connection covered by the given requirement.

7. Concluding remarks

We have presented a simple yet flexible approach for ensuring thread-safety for composite object systems. Our model requires knowledge of the exclusion requirements on
the interfaces of base-level components, the dependency of the interface of each composite object on its components, and knowledge of what the potential concurrent activation of the system might be in its operating environment. We rely on a Galois connection between the outward mapping of exclusion requirements, and the inward mapping of potential concurrency, to reduce the locks considered per component to a minimal subset of those possible. Given the choice of a particular distribution of locks throughout the components of the system, we can calculate whether or not each component is indeed thread-safe, where locks are redundant and where high level or coarse grain locks cause potential for concurrency to be lost.

A potential criticism of our approach is that it is only relevant for object systems with encapsulated components. In fact, the same ideas apply more generally. Because our model is phrased quite abstractly, we can choose to deal with any controllable program entity at all (for example, particular critical sections of code, or even read or write access on individual program variables). We have merely presented our approach using object methods and interfaces as a vehicle for the ideas. The key issue is that we must be able to capture the uses dependency relationship for those entities that we wish to control. Another limitation of our approach is that it does not consider 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.

Our model presumes 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 [11,10,9], 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.

Appendix A. An algebra of exclusion

A.1. Basic definitions

The form of expressions is given in Fig. A.1 with operators listed from high to low precedence. Disjunction has two distinct forms, with the higher precedence form using concatenation, which is convenient for listing the elements of a set. An expression \( e \) denotes an undirected graph with a vertex set \( \mathcal{N}_e \subseteq \text{NAME} \) and a set of edges \( \mathcal{P}_e \), as defined in Fig. A.2.

The key feature of the algebra of exclusion, in comparison with conventional relational algebra, is that the carrier set \( \mathcal{N} \) is part of the meaning of an expression, and is not taken as a given universal set. We can combine expressions with different carriers, and more importantly distinguish expressions denoting the same set of pairs \( \mathcal{P} \), but with different vertex sets. This distinction is important when we wish to convey the scope of an exclusion requirement. In effect, the vertex set specifies a component interface, and the edge set specifies the exclusion requirement or concurrency potential for that interface.

In Fig. A.2, the null expression \( 0 \) denotes an empty graph, with empty vertex set, and the name \( n \) a graph with a single vertex \( n \) and empty edge set. A disjunction \( e_1 \mid e_2 \) forms the union of two graphs by taking the union of their vertex sets and of their edge
### Syntax of Expressions

<table>
<thead>
<tr>
<th>EXP e</th>
<th>N_e</th>
<th>P_e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>n</td>
<td>{n}</td>
<td>∅</td>
</tr>
<tr>
<td>e_1 \ e_2</td>
<td>N_{e_1} \cup N_{e_2}</td>
<td>P_{e_1} \cup P_{e_2}</td>
</tr>
<tr>
<td>e_1 \times e_2</td>
<td>N_{e_1} \times N_{e_2}</td>
<td>P_{e_1} \cup P_{e_2} \cup (N_{e_1} \times N_{e_2}) \cup (N_{e_2} \times N_{e_1})</td>
</tr>
<tr>
<td>\overline{e}</td>
<td>N_e</td>
<td>∅</td>
</tr>
<tr>
<td>e_1 - e_2</td>
<td>N_{e_1}</td>
<td>P_{e_1} \cap P_{e_2}</td>
</tr>
<tr>
<td>e^c</td>
<td>N_e</td>
<td>(N_e \times N_e) - P_e</td>
</tr>
</tbody>
</table>

\[
\llbracket e \rrbracket \equiv (N_e, P_e)
\]

\[\begin{align*}
\mathcal{N} &: \text{EXP} \rightarrow \mathbb{P} \text{NAME} \\
\mathcal{P} &: \text{EXP} \rightarrow \mathbb{P} (\text{NAME} \times \text{NAME})
\end{align*}\]

### Semantics of Expressions

Sets. The product form is the most unconventional part of our algebra. A product \(e_1 \times e_2\) combines the symmetric cartesian product \((N_{e_1} \times N_{e_2}) \cup (N_{e_2} \times N_{e_1})\) of the respective vertex sets, which is the primitive means for introducing new pairs, with the existing edge sets \(P_{e_1} \cup P_{e_2}\). Completion \(\overline{e}\) represents the complete graph on the names of \(e\). Completion is an abbreviated product: \(\overline{e} = e \times e\). Support \(g\) simply represents the set of names of \(e\) with no pairs; it is equivalent to the complement of the completion: \(g = \overline{\overline{e}}\). A difference \(e_1 - e_2\) removes any pairs of the second expression from the first, leaving its vertex set as is. Complement is not taken with respect to some complete universal graph, but is relative to the expression being complemented. Complement is an abbreviated difference: \(e^c = \overline{e} - e\).

Expressions can be partially ordered in the obvious way, via

\[e_1 \subseteq e_2 \text{ iff } N_{e_1} \subseteq N_{e_2} \text{ and } P_{e_1} \subseteq P_{e_2}.
\]

Although not used here, we can make an extension of our algebra with other operators, such as conjunction \(e_1 \& e_2\) and restriction \(e_1 \setminus e_2\). Conjunction is graph intersection, and restriction is similar to difference, but the vertex set, for both conjunction and restriction, is the intersection of the operands’ vertex sets. We find \(e_1 \setminus e_2 = e_1 \& e_2^c\), but de Morgan’s
Some properties of the algebra of exclusion are listed in Fig. A.3. The validity of these properties follows directly from the semantics of Fig. A.2. Disjunction | and product × are commutative, associative with identity element 0. Disjunction distributes through product, and vice versa. Disjunction is idempotent. All operators enjoy monotonicity properties: disjunction, product, completion and support are all order-preserving in all arguments, as is difference in its first argument; difference is order-reversing in its second argument. The relativised complement operator is only order-reversing when restricted to expressions with the same support. It is obvious from the semantics that any expression has an equivalent canonical form, unique up to listing order, comprising a disjunction of all the name-pairs in the expression, together with a disjunction of all the remaining unpaired names in the support of the expression. However, the canonical form yields overly long expressions. The syntax has been designed to be convenient for exclusion expressions; for example, collections of read–write sets can be written compactly.

The algebra is a distributive lattice with disjunction and conjunction as the join and meet operators. The sublattice of expressions e restricted to the interval $\epsilon \subseteq e \subseteq \epsilon^c$ is simply the Boolean algebra of symmetric binary relations on the finite set of names of $\epsilon$. We have deliberately constrained the notation to be finitary—we can only model finite graphs. For example, even if the underlying set NAME is infinite, we have no way of using more than a finite number of names in an expression; in particular we have refrained from defining a universal complement which would have extended our model to co-finite graphs, and given us a Boolean algebra with de Morgan laws.

The novel operator in the algebra of exclusion is the product operator. There are some properties of duality between disjunction and product that are convenient for manual calculation of complements in examples.

*Duality property for | and ×:* Given disjoint $e_1$ and $e_2$ (that is, $e_1$ and $e_2$ have no names in common: $e_1 \& e_2 = 0$), we have the following rules for forming complements, which may be recursively applied to subexpressions:

\[
(e_1 \mid e_2)^c = e_1^c \times e_2^c \quad (e_1 \times e_2)^c = e_1^c \mid e_2^c.
\]

\[
\begin{align*}
e_1 \mid e_2 & = e_2 \mid e_1 = e_1 e_2 \\
e_1 \times e_2 & = e_2 \times e_1 \\
e_1 \mid e_2 \times e_3 & = e_1 e_2 \times e_1 e_3 \\
e_1 \times e_2 \times e_3 & = e_1 \times e_2 \times e_1 \times e_3 \\
e & = 0 = e \quad e = e \times 0 = e^{c,c} \\
\overline{e} = e \times e & = e \mid e^{c} = \overline{e} = \overline{e^c} \\
\overline{e_1 e_2} = e_1 \mid e_2 & = e_1 \times e_2 = \overline{e_1} \times \overline{e_2} \\
0 & = \overline{0} = 0^c \\
0 \subseteq e_1 & \subseteq e_1 \mid e_2 \subseteq e_1 \times e_2
\end{align*}
\]
Typically, the recursive complementation terminates with

\[ \overline{\epsilon^c} = \epsilon \quad \epsilon^c = \overline{\epsilon}. \]

For example, consider the calculation of \( P_I \) in the GUI server:

\[ P_I = \overline{\overline{mdc l_i l_o}} - mdc \times l_i l_o \]

[from Fig. 14]

\[ = (mdc \times l_i l_o)^c \]

[by definition of complement]

\[ = (mdc)^c \mid l_i l_o \]

[by duality property of complement]

\[ = \overline{mdc} \mid l_i l_o \]

[ditto]

\[ = mdc l_i l_o \]

[by property of support].

References