Flexible Concurrency Control Granularity through Nested Encapsulation in Hermes/ST *

Michael Fazzolare
Telecommunications Software Research Centre (TSRC)
The University of Wollongong, Department of Computer Science
P.O. Box 1144, Wollongong NSW 2522, Australia
Tel: (+61 42) 26 8832, Fax: (+61 42) 27 3277,
Email: faz@cs.uow.edu.au

Abstract

The Hermes/ST distributed programming environment supports the creation of reliable applications through the use of nested transactions. An important tradeoff in any transactional system is the amount of parallelism desired versus the cost of providing that parallelism. Hermes/ST allows applications to balance the tradeoff in two novel ways. Static multi-granular concurrency control allows *classes* to be instantiated with different granularities of concurrency control. Dynamic multi-granular concurrency control allows *instances* to vary their level of concurrency control during an application's execution. Both forms of multi-granular concurrency control are implemented over a nested encapsulation object topology. Hermes/St also supports traditional intention locking. Multi-granular concurrency control and intention locking are compared and contrasted.

1 Introduction

Pessimistic concurrency control with strict two phase locking is a common way to ensure serialisability in transactional systems [BHG87]. Distributed transactional systems need to allow developers to tradeoff the amount of parallelism against the overheads to supply a given level of parallelism. High parallelism is particularly desirable in a transactional system with strict two phase locking because objects are locked until a transaction commits or aborts. However high parallelism incurs a high execution time locking overhead that may adversely affect transaction throughput. Low parallelism has low execution overhead but may cause long transaction response time and low throughput.

Distributed transactional systems usually approach this problem by forcing the application developer to hard code the level of parallelism in the implementation of an abstract data type. But often, different instances of a particular abstract data type require different levels of parallelism.

Hermes/ST [FHR93b, FHR94][1] is a distributed programming environment that supports the construction of reliable distributed applications that utilise *nested transactions* [Mos85]. Through the introduction of a *nested encapsulation model* that restricts object visibility, Hermes/ST has introduced the ability for the application developer to specify the level of parallelism based on execution time factors. Hermes/ST defines concurrency controllers over individual variables, individual objects and over groups of objects.

*This research was funded by Telecom Australia under Contract No. 7260.

[1] Not to be confused with IBM's Hermes/ST System

It further defines an ordering for these concurrency controllers that in combination with restricted object visibility allows flexible and reusable concurrency control for applications.

Two forms of concurrency control are permitted. *Static multi-granular concurrency control* allows the developer to specify a fixed level of parallelism for particular instances of an abstract data type. It also allows the reuse of existing applications with different concurrency control granularity with minimal change to the implementation and no change to concurrency control specifications. *Dynamic multi-granular concurrency control* allows instances of abstract data types that can *vary* their level of concurrency control during runtime according to factors signalled by *triggers*. *Down triggers* cause more parallelism whereas *up triggers* cause less parallelism. Triggers have a default behaviour which can be redefined to suit the needs of individual applications.

This paper is structured as follows. Section 2 describes the Hermes/ST distributed object model, including its nested encapsulation structure. Section 3 discusses the motivation for static and dynamic multi-granular concurrency control. Section 4 presents static multi-granular concurrency control, which can be viewed as a special case of dynamic multi-granular concurrency control presented in Section 5. Section 6 compares the work with other approaches to concurrency control specification in distributed transactional systems. Section 7 introduces intention locking. Section 8 discusses the difference between multi-granular concurrency control and intention locking. Finally, Section 9 presents conclusions.

2 The Hermes/ST Distributed Object Model

Hermes/ST allows system developers to construct applications using distributed objects. It incorporates a novel method invocation scheme that supports both transactional and non-transactional threads [Hum93]. This section describes only those portions of the object model that are relevant to this paper.

The Hermes/ST distributed object model consists of *Hermes/ST objects*. Hermes/ST objects encapsulate variables. These variables are used to *refer* to other Hermes/ST objects. Hermes/ST objects export public interfaces (methods) that allow interaction with other Hermes/ST objects. Hermes/ST objects can be *constant* which means that their value is immutable [Atk91]. Hermes/ST objects can be *transactional* which means they are *concurrency controlled*, have *state-restoration* facilities and are *persistent* [BHG87]. Hermes/ST objects can be *volatile* which means they are neither constant nor transactional.

Hermes/ST objects interact via *message passing* [GR89]. Hermes/ST interface methods are invoked with parameters and return results. Parameters and results can be constant, volatile or transactional Hermes/ST objects. Constant and volatile objects are passed *by value*. Hermes/ST objects are passed *by reference*.

The Hermes/ST transactional thread model permits methods to be invoked transactionally. It further allows these method invocations to be either *synchronous* or *asynchronous*. Asynchronous method invocations proceed independently, whereas synchronous method invocations cause the invoker to be suspended until the method invocation has completed. Nested transactional method invocations cause nested transactions to be constructed. In this paper, only *transactional threads* and are discussed. Transactional threads is the collective term used to refer to any synchronous or asynchronous, subtransaction creating or not, method invocations made from within a transaction.

2.1 Nested encapsulation

Hermes/ST extends the traditional object–variable encapsulation structure so that Hermes/ST objects can encapsulate other Hermes/ST objects. A Hermes/ST object is either *contained* by the Hermes/ST

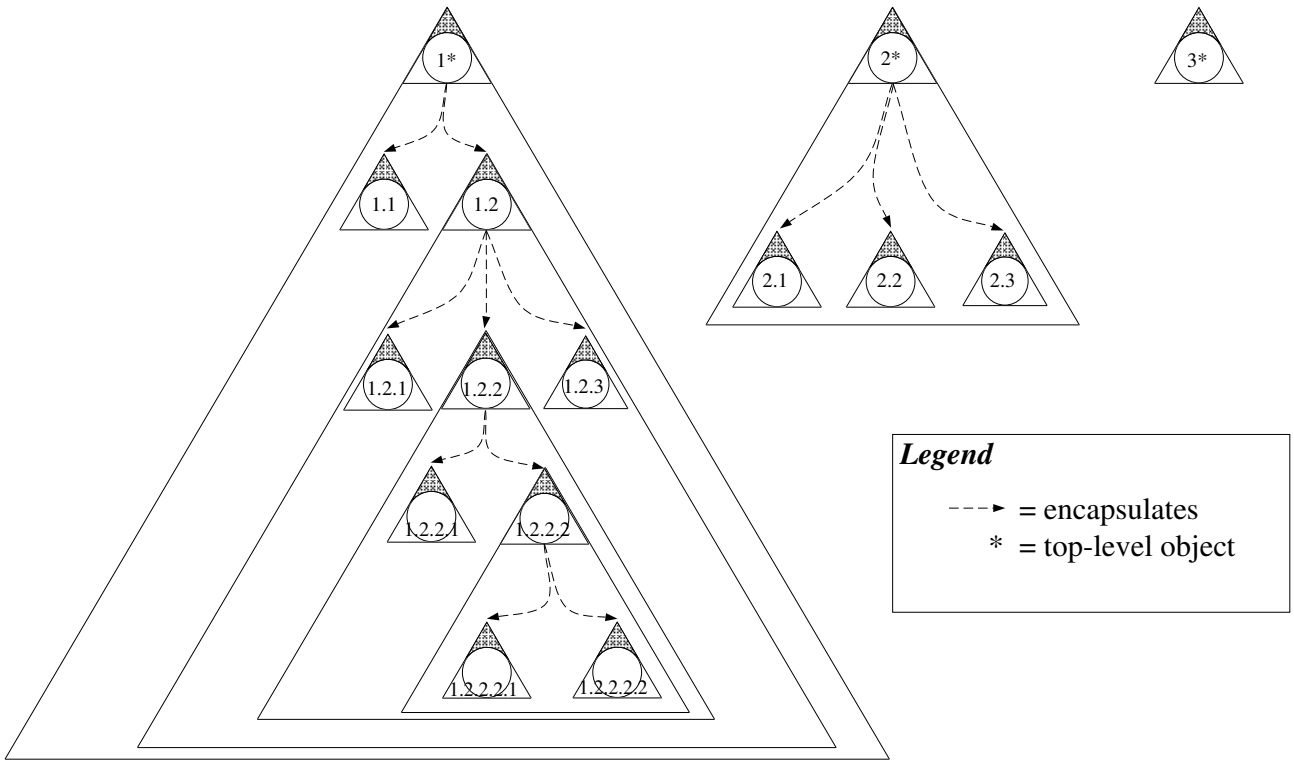

Figure 1: Example of a encapsulation hierarchy. Circles represent Hermes/ST objects. Triangles represent encapsulation boundaries. The shaded part of a triangle represents an object's public interface. Objects O_1, O_2 and O_3 are top-level.

system or by another Hermes/ST object. Hermes/ST objects contained by the system are called *top-level objects*. A Hermes/ST object and all of its contained descendants is known by the collective term *encapsulating object*.

The containment hierarchy formed through nested encapsulation produces a tree structure for each top-level encapsulating object. Structuring the encapsulation hierarchy as a tree means that it is not possible for a Hermes/ST object to be contained by more than one Hermes/ST object. This is consistent with the manner in which objects encapsulate their variables. A Hermes/ST object can only be contained by one parent, just as a variable can only be contained by one object[2]. A Hermes/ST object's encapsulating parent is specified during its creation and the "is contained by" relationship is maintained during the life of the Hermes/ST object.

A Hermes/ST object's position in the encapsulation hierarchy is used to uniquely name the object. These *name paths* are generated automatically by Hermes/ST during Hermes/ST object creation. More convenient user-defined *aliases* are available via the Hermes/ST name server [FHR93b].

Figure 1 illustrates the Hermes/ST nested encapsulation structure. Encapsulation boundaries are represented as triangles. Hermes/ST objects are named so as to reflect their position in the containment hierarchy[3]. Thus $O_{1.2.3}$ is contained by $O_{1.2}$ which is further contained by O_1 which is a top–level object.

Just as the encapsulation of variables by objects in an object–oriented language restricts the visibility

[2]Of course, as many objects as necessary may possess a reference to a Hermes/ST object.

[3]The components of a name path used in this paper have been simplified to integers for illustrative purposes. The actual Hermes/ST name paths uniquely specify the object through a concatenation of node name and sequence number.

of variables, so too does Hermes/ST's nested encapsulation structure restrict the visibility of Hermes/ST objects. With respect to illustrations such as Figure 1 the encapsulated object *access rule* can be described as:

> *An object may communicate with the interface of any object that it can reach by* leaving *encapsulation boundaries (the triangles in Figure 1) only, never by entering them.*

The access rule captures the intuitive understanding of what encapsulation means. More concretely, a Hermes/ST object (say $O_{1.2.2}$) is only allowed to communicate with:

- its direct children and siblings ($O_{1.2.1.1}$, $O_{1.2.2.2}$ and $O_{1.2.1}$, $O_{1.2.3}$).
- its parent and its parent's siblings (uncles) ($O_{1.2}$ and $O_{1.1}$).
- its grandparent and its grandparent's siblings (great uncles) (O_1 and O_2, O_3).
- and so on...

Conversely, it ($O_{1.2.2}$) cannot talk to:

- its grand children or deeper descendants ($O_{1.2.2.2.1}$, $O_{1.2.2.2.2}$).
- its cousins including first, second ...cousins ($O_{2.1}$, $O_{2.2}$, $O_{2.3}$).
- and so on...

As a final note it is important to mention that the restricted access imposed by the Hermes/ST nested encapsulation structure still allows arbitrary Hermes/ST object communication topologies. Trivially, all top-level objects are siblings and thus can form topologies where each object can communicate with every other object. Similarly, all objects at one level of the containment hierarchy can communicate in an arbitrary fashion.

3 Granularity Issues

The object model, as introduced in the previous section, allows Hermes/ST application developers to provide object structures for their programs. Transactional objects provide concurrency control, state restoration and persistence handling features. Concurrency control is needed in transactional systems to maintain *serialisability* of transaction schedules [BHG87]. This paper is only concerned with the provision of concurrency control for transactional objects. State restoration and persistence handling are detailed in [FHR93b]. More specifically this paper is concerned with the provision of appropriate granularity of concurrency controllers for Hermes/ST transactional objects.

The goal of concurrency control is to provide high throughput and low response times for transactions. Selecting the correct level of concurrency control to maximise throughput and minimise response time involves a tradeoff between high concurrency and the overheads incurred in providing this concurrency [BHG87]. Existing approaches (see section 6) tie the level of granularity to the implementation of a particular abstract data type. Hermes/ST approaches the tradeoff in two ways. Firstly classes that implement abstract data types are not necessarily forced to have a fixed granularity. Thus a class is instantiable with different granularities. Secondly, individual instances of classes should not necessarily have a fixed granularity. Thus, instances of classes implementing abstract data types are allowed to exhibit different levels of concurrency control granularities at different times during their usage.

Application developers can define the appropriate level of class or instance concurrency control granularity at the following times:

1. During the *development* of the abstract data type. The idea here is that the application developer can construct an implementation of an abstract data type and test several different concurrency control granularities on this implementation. The most suitable concurrency control granularity can then be chosen for a completed application. This approach is supported in Hermes/ST because a class can be instantiated with different concurrency control granularities.

2. *Reuse* of an abstract data type implementation. The idea here is that there will be a need to utilise the same implementation with different concurrency control granularities. Thus a class representing a searchable collection, might be instantiated with high parallelism in a distributed directory application. The *same* class might be instantiated with low parallelism in an application that maintains local user profiles. As above, this approach is supported because different instances can have different concurrency control granularities.

3. During the *use* of the abstract data type. The motivation for this type of granularity of concurrency control is that objects may benefit from varying their granularity depending on information only available during execution. This information could be, for example, related to the recent access patterns of the object structure, or could be based on the time of the day. This approach is supported because individual instances can exhibit different levels of concurrency control at different points in time.

4 Static Multi-Granular Concurrency Control

Establishing a fixed granularity of concurrency control for individual *instances* of object structures is called *static multi-granular concurrency control* in Hermes/ST. Hermes/ST facilitates static multi-granular concurrency control by defining three types of concurrency controllers.

As–specified concurrency controller: An *as–specified* concurrency controller is so named because it means to use one of two types of high parallelism concurrency controllers "as the user specified". The two types of high parallelism concurrency controllers defined by Hermes/ST are:

Implicit Concurrency Controller: An implicitly concurrency controlled Hermes/ST object has one concurrency controller for each of its variables. All concurrency control is done automatically when variables are accessed (read or written). The advantages and disadvantages of implicit concurrency control are discussed in [FHR94, FHR93a].

Explicit Concurrency Controller: An explicitly concurrency controlled Hermes/ST object has one concurrency controller that controls access to the entire Hermes/ST object. Individual method invocations have user-definable locks associated with them. These locks are instantiated during method invocation, and their compatibility with other granted locks is used to schedule their execution. Compatible methods (as defined by the developer) are executed immediately; incompatible methods are delayed. The advantages and disadvantages of explicit concurrency control are discussed in [FHR94, FHR93a].

High parallelism Hermes/ST objects that have either implicit or explicit concurrency controllers are henceforth called *as–specified objects*.

Mutual Exclusion Concurrency Controller: A mutual exclusion concurrency controlled object also has one concurrency controller that synchronises access to an entire Hermes/ST object. However, a mutual exclusion concurrency controller only allows one transaction to be active in an object at

any one time. The restricted object visibility imposed by nested encapsulation implies that there can only be one transaction in the entire encapsulated object associated with the mutual exclusion concurrency controller. This means that descendants of an mutual exclusion concurrency controlled object need not have any concurrency control.

Hermes/ST allows an application developer to specify the position of a mutual exclusion concurrency controller in an application's encapsulation hierarchy. This is because the type of concurrency controller that is to be associated with a Hermes/ST object is specified during its creation. This allows the granularity of concurrency control to be dynamic over classes.

The application developer can replace the system defined mutual exclusion concurrency controller with a user defined mutual exclusion concurrency controller if such an action is so desired. This could be done to permit more than one transaction to be active in an encapsulated mutual exclusion object. It is the developer's responsibility to ensure that serialisability is maintained in such cases [4].

Hermes/ST objects that have a mutual exclusion concurrency controller are henceforth referred to as *mutex objects*.

No Concurrency Controller: As was motivated by mutual exclusion concurrency controllers, some Hermes/ST objects do not need concurrency controllers at all. The encapsulation structure ensures that descendants of a mutex object can only have one transaction active in them at a time. Objects with no concurrency control, by definition do not have locking overhead associated with them.

Hermes/ST objects that have no concurrency controllers are henceforth referred to as *none objects*.

4.1 The Order Rule

Hermes/ST's three types of concurrency controllers allow both extremes of the concurrency control granularity tradeoff to be employed by an application. As–specified objects allow data structures that support high concurrency to be developed. Mutex objects and their consequential descendant none objects allow low overhead data structures to be developed.

For some data types it is convenient to allow a mixture of high parallelism and low overhead concurrency control. In a data structure such as a Btree it could be desirable to allow high concurrency at the root node (because every operation must access it) and low concurrency at leaf nodes (because they are accessed relatively infrequently). To allow such behaviour to be implemented, Hermes/ST defines a rule governing the positioning of various types of concurrency controllers in an encapsulated structure. The concurrency controlled object *order rule* is:

> *An individual path in the encapsulation tree can contain zero or more as–specified objects. It can optionally be followed by one mutex object, which can be followed by zero or more none objects.*

The rule allows parallelism above a mutex object and no parallelism at or below a mutex object. This order is both desirable and necessary.

The rule is *necessary* because arbitrary mixtures of as–specified, mutex and none objects could violate serialisability. An as–specified object having a none object as a child is the case in point. The as–specified parent object can have conflicting transactions accessing different parts of itself. These transactions are

[4] In order to make later discussions concise, user and system defined mutual exclusion concurrency controlled objects are collectively referred to as mutex objects.

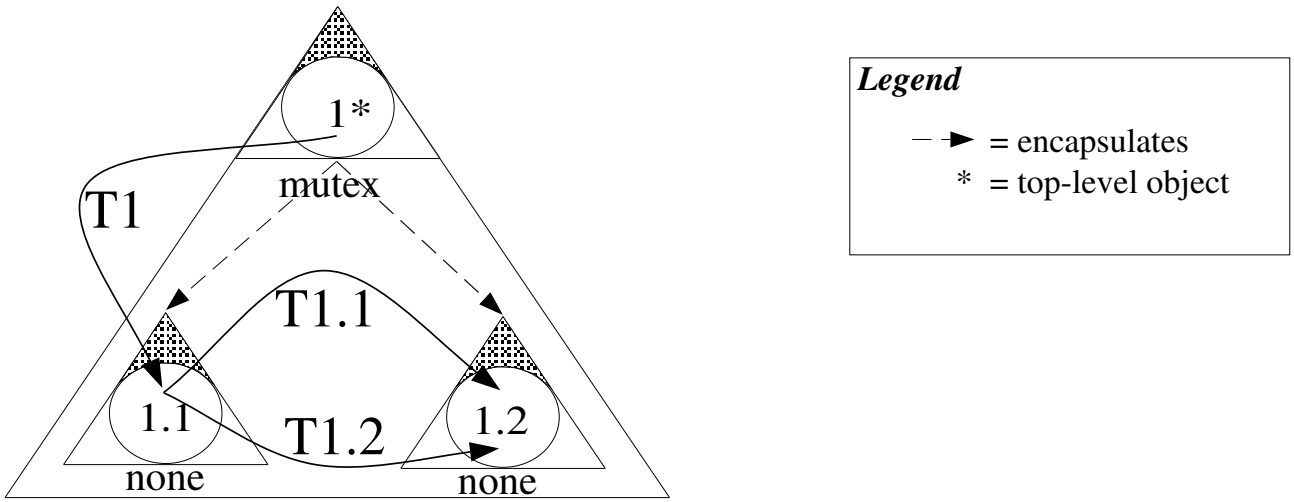

Figure 2: Example of an asynchronous communication that violates serialisability. $O_{1.1}$ performs two asynchronous method invocations on $O_{1.2}$, which has no concurrency controller defined.

allowed to communicate with the child none object. Thus the parent as-specified object could allow both conflicting transactions to become active in the child, defying serialisability.

The order rule is *desirable* because any concurrency controllers below a mutex object would simply be wasted. Sequencing violations of the order rule (e.g. mutex object having a child mutex object) do not invalidate serialisability and are therefore relatively harmless. They are referred to as *wasteful orderings*.

4.2 The Sneaking Thread Problem

Even with the order rule it is still possible to violate serialisability *within* a Hermes/ST encapsulating object. Recall from Section 2 that asynchronous or synchronous transactional threads can be generated during Hermes/ST object method invocations. Further recall from Section 2 that the encapsulation access rules allow Hermes/ST objects to communicate with children and siblings.

Figure 2 illustrates this problem. Even though the parent mutex object O_1 allows only *one* transaction (T_1) to be active in its encapsulated subtree. the none object $O_{1.1}$ can generate two nested asynchronous subtransactions ($T_{1.1}$, $T_{1.2}$) at $O_{1.2}$. These two transactions are not concurrency controlled, because $O_{1.2}$ is a none object. This may violate serialisability, and is referred to as the *sneaking–thread* problem.

This problem can only occur *within* an encapsulating mutex object. The reason for this is that a none object cannot be a top–level object (order rule) and consequently is *invisible* from outside the encapsulating mutex object (access rules).

To solve the sneaking–thread problem, Hermes/ST serialises[5] all asynchronous transactional method invocations to none objects. This serialisation imposes an artificial execution order on the asynchronous subtransactions. However, since the subtransactions were specified to be asynchronous, they were to be serialised in some convenient system order anyway. The "convenient system order" now preferred by Hermes/ST is "one after the other". Obviously, the subtransactions now exhibit less parallelism. This may affect their execution speed, but does not alter the application semantics. Finally, deadlock cannot

[5]For systems such as Argus [Lis88] that suspend the parent transaction while executing a subtransaction, this can be achieved by converting the asynchronous call into a synchronous one. In Hermes/ST where the parent transaction can execute in parallel with the child subtransaction the serialisation is more complex [Hum93].

be introduced because the conversion to a synchronous schedule does not introduce any new cycles in the *waits–for graph* [BHG87].

In order to implement the solution to sneaking-thread problem, Hermes/ST objects have to be able to determine when they are communicating with a none object. As–specified objects know that any objects they can access are guaranteed to be as–specified or mutex objects (order rule) and therefore need no special treatment.

Mutex objects know that all objects accessible *below* them are always none objects (order rule) and therefore need asynchronous–synchronous invocation conversions. Objects accessible *above* mutex object are always as–specified or mutex (order rule) and need no special treatment.

None objects know that all accessible objects *below* them are always none objects (order rule) and therefore need asynchronous–synchronous invocation conversions. Objects accessible above the none object but below the mutex object are none objects (order rule) and also need invocation conversions. Objects at or above the mutex object and in the same path can only be as–specified or mutex objects (order rule) and therefore need no special treatment. Objects above the mutex object and *not in* the same path that are visible (access rules), can only be as–specified objects or children of as–specified objects (mutex or as–specified) (order rule). These objects need no special treatment. Thus none objects need only keep track of their single ancestor mutex object to deal with the sneaking–thread problem[6].

5 Dynamic Multi-Granular Concurrency Control

As mentioned in Section 3, there are times when an application developer may wish to vary the granularity of concurrency control of an *individual* instance of an abstract data type depending on information only available during execution.

To facilitate the specification of different levels of concurrency control granularity, Hermes/ST introduces *highest* and *lowest* markers. The markers section each encapsulation hierarchy into three regions.

1. In the region above the highest marker, Hermes/ST objects will always be concurrency controlled as the programmer specified them. This region always allows parallelism.

2. From the highest marker to the lowest marker represents the region in which the system is free to determine the most suitable level of parallelism supplied. Naturally, the system must comply with the order rule in choosing the type of concurrency controllers that will achieve a certain level of parallelism. Higher parallelism is achieved by moving mutex objects down the tree. Lower parallelism is achieved by moving mutex locations higher in the tree.

3. In the region below the lowest marker, all objects do not have a concurrency controller. There is no parallelism in this region.

When highest and lowest markers are always positioned together or they are omitted entirely, dynamic multi-granular concurrency control reduces to static multi-granular concurrency control. Encapsulating top–level objects O_1, O_2 and O_3 illustrate different examples of this. (see Figure 3). Encapsulating object O_1, where neither lowest nor highest markers have been specified produces a fixed containment hierarchy that supports high concurrency. Encapsulating object O_2, which has a highest and a lowest marker positioned at the top-level object O_2, produces a fixed containment hierarchy that has no parallelism. Encapsulating object O_3, which has highest and lowest markers positioned at two contained objects ($O_{3.1.1}$ and $O_{3.2}$), produces the most general sort of static multi-granular concurrency control hierarchy,

[6]Section 5 introduces movement of the mutex object, hence the term "keep track".

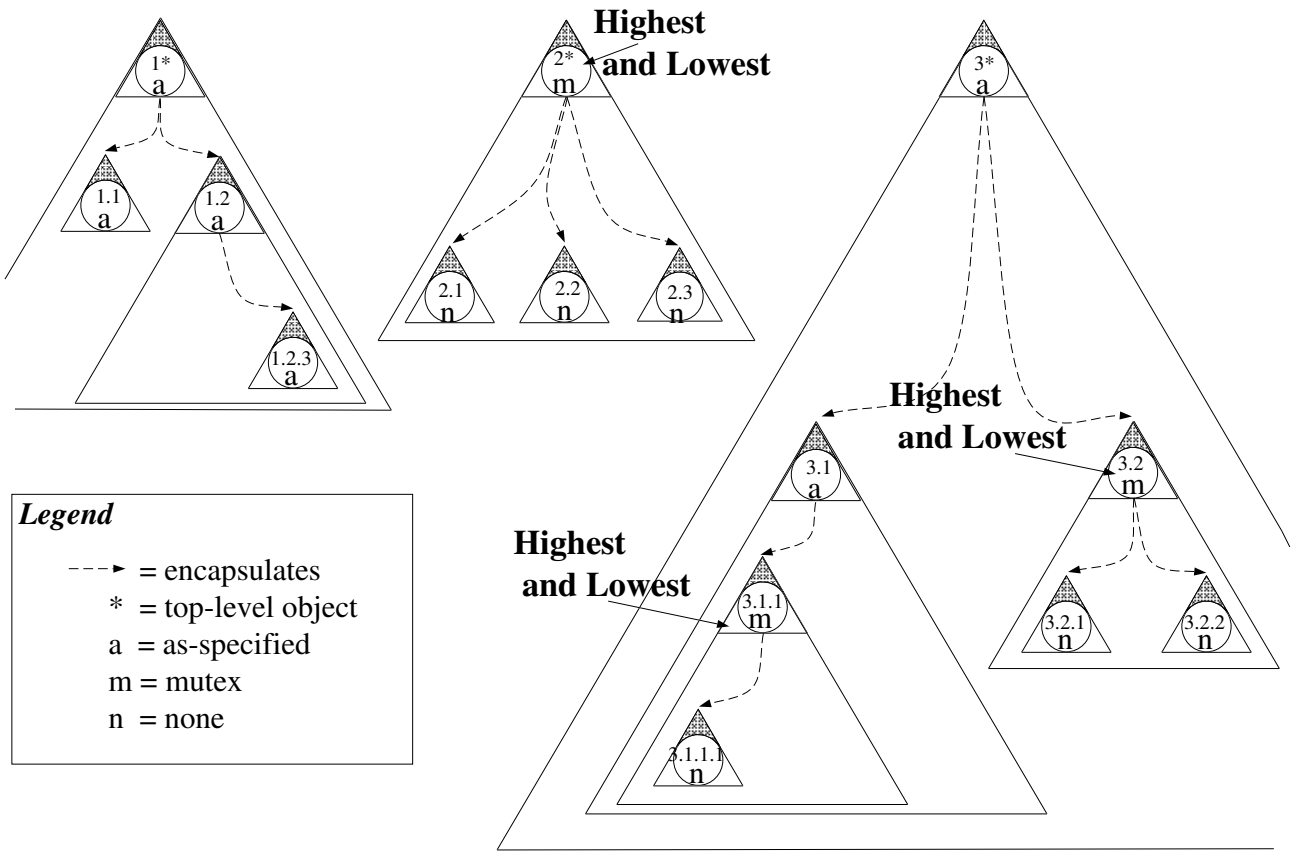

Figure 3: An example encapsulation hierarchy showing the various styles of (static) multi-granular concurrency control that Hermes/ST supports by specifying that highest and lowest markers be positioned together.

i.e. one in which there is high parallelism followed by no parallelism at fixed levels in the encapsulation hierarchy.

The encapsulating top–level object O_4 (see Figure 4) illustrates a containment hierarchy over which the system is entirely free to determine the appropriate level of granularity. Encapsulating object O_4 has the highest marker positioned at itself and has omitted specifying a lowest marker. This indicates that all objects below O_4 in the contained hierarchy can be one of as–specified, mutex or none. The illustration depicts one possible scenario, where the mutex objects exist at $O_{4.1.1}$ and $O_{4.2}$. There are many other possible configurations for the encapsulating object O_4. Naturally, all configurations must comply with the order rule.

Encapsulating top–level object O_5 (see Figure 5) shows the most general form of dynamic multi-granular concurrency control. This kind of dynamic multi-granular concurrency control has highest and lowest markers that are positioned at *different* Hermes/ST objects. It has Hermes/ST objects that are above a highest marker and are always concurrency controlled as the user specified (O_5, $O_{5.1}$, $O_{5.1.2}$). It has Hermes/ST objects that contain a highest marker, which can be as–specified or mutex ($O_{5.1.1}$, $O_{5.2}$). It has Hermes/ST objects that are below a highest marker and above a lowest marker that can be as–specified or mutex or none ($O_{5.2.1}$, $O_{5.2.2}$, $O_{5.2.3}$, $O_{5.2.1.1}$, $O_{5.2.3.1}$, $O_{5.2.3.2}$). It also has a Hermes/ST

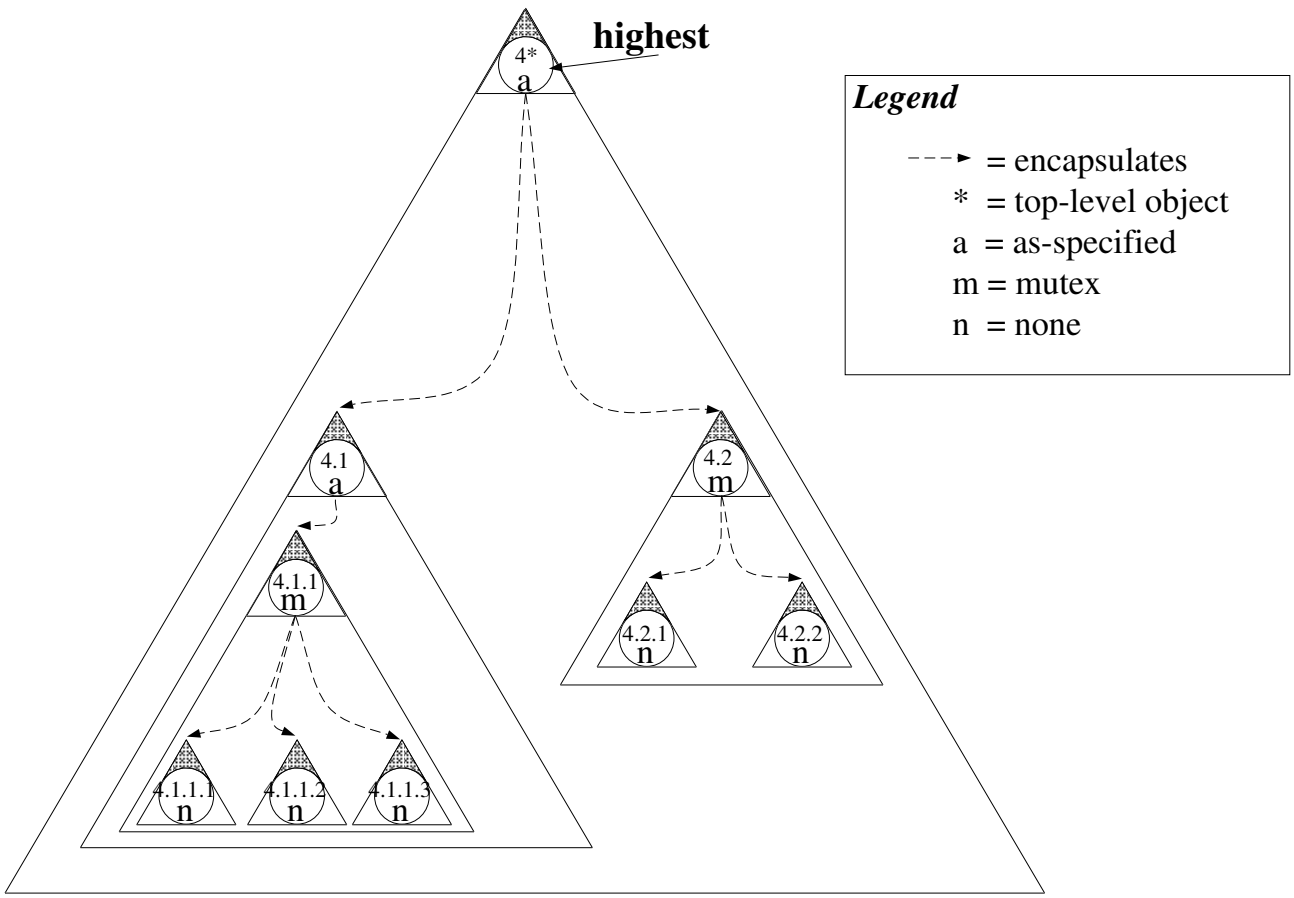

Figure 4: An example of dynamic granularity concurrency control showing the highest marker at the top-level Hermes/ST object.

object[7] that contains the lowest marker and hence can be either mutex or none ($O_{5.2.1.2}$). Finally, it has Hermes/ST objects that are below the lowest marker and that must be a none object ($O_{5.2.1.2.1}$, $O_{5.2.1.2.2}$).

5.1 Triggers

When the highest and lowest markers are positioned at different Hermes/ST objects in a containment hierarchy, Hermes/ST can change the level of parallelism exhibited by the hierarchy. This process is referred to as *concurrency control movement*. Moving concurrency controllers *up*, in an encapsulation hierarchy decreases parallelism in that hierarchy. Moving concurrency controllers *down* in an encapsulation hierarchy increases parallelism in that hierarchy. Concurrency control movement is initiated by *trigger objects*. *Up triggers* initiate upward concurrency controller movement, whereas *down triggers* initiate downward concurrency controller movement.

Down triggers are associated with each mutex object. Up triggers are associated with any as–specified objects that exist between the highest marker and the current mutex object along any given path. If there are multiple up triggers requests in the same encapsulating sub-tree then only the highest trigger is serviced. If there are conflicting requests for an object to be involved in both down and up concurrency control movement, then the up trigger movement is chosen.

Trigger objects have a system default behaviour that tries to maximise throughput and minimise response time of transactions according to the *general granularity curve* [BHG87]. Users may redefine a trigger object's *trigger method* so that concurrency controller movement is initiated according to type specific traffic conditions, or other external events such as the time of the day.

5.2 Downward Concurrency Controller Movement

Figure 6 (a) illustrates an example encapsulation hierarchy before it has moved its concurrency controllers down. The hierarchy consists of a single mutex object (O_1) and its associated down trigger encapsulating a sub-hierarchy of none objects ($O_{1.1}$, $O_{1.2}$, $O_{1.2.1}$, $O_{1.2.2}$). Figure 6 (b) shows the same example encapsulation hierarchy after it has moved its concurrency controllers down. The former mutex (O_1) object has become an as–specified object and all encapsulated children ($O_{1.1}, O_{1.2}$) have become mutex objects. More deeply encapsulated descendants ($O_{1.2.1}, O_{1.2.2}$) have remained none objects.

Downward concurrency controller movement can be viewed as moving the mutex object down a level in the encapsulation hierarchy. The algorithm to do this can be expressed in the following manner:

```
1. If there is a transaction in the down triggerring
   mutex object then introduce ''restricted concurrency''
   until it has committed. Delay all other transactions.
2. Insist [all children become mutex objects].
3. Change the down triggerring mutex object
   to an as-specified object.
4. Inform all descendants of the position of their
   new mutex parent.
5. Re-schedule any delayed transactions from step 1.
```

Before the move down algorithm can be performed, any transactional thread[8] active in the encapsulated hierarchy of the down triggerring object must commit. This is accomplished by Step 1 through

[7] In this example there is only one lowest marker specified. There can, of course, be more than one lowest marker in an encapsulation hierarchy.

[8] As the down triggering object is a mutex object, there can only be one active transaction (possibly nested).

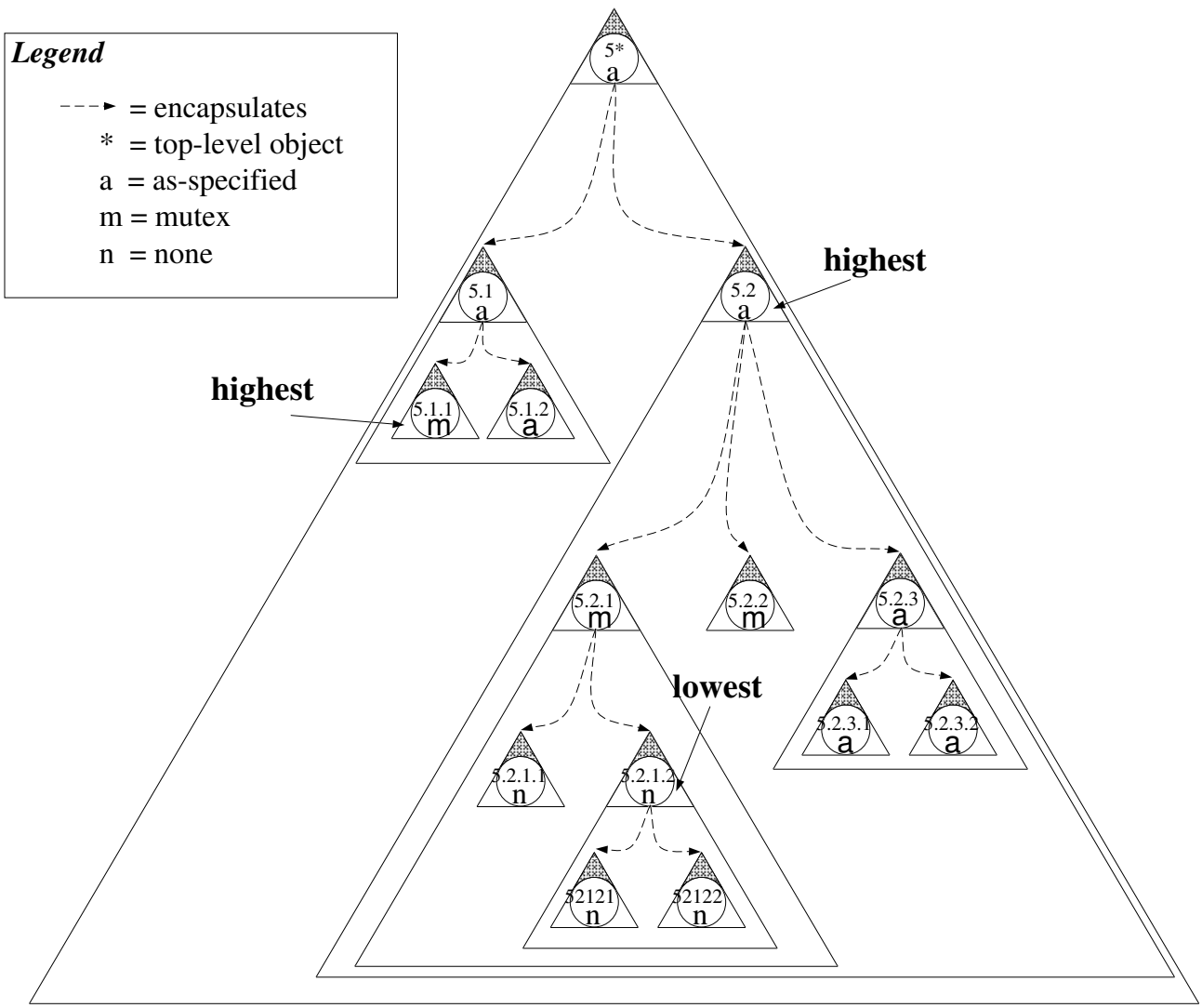

Figure 5: An example encapsulation hierarchy showing the most general form of dynamic concurrency control granularity that Hermes/ST supports.

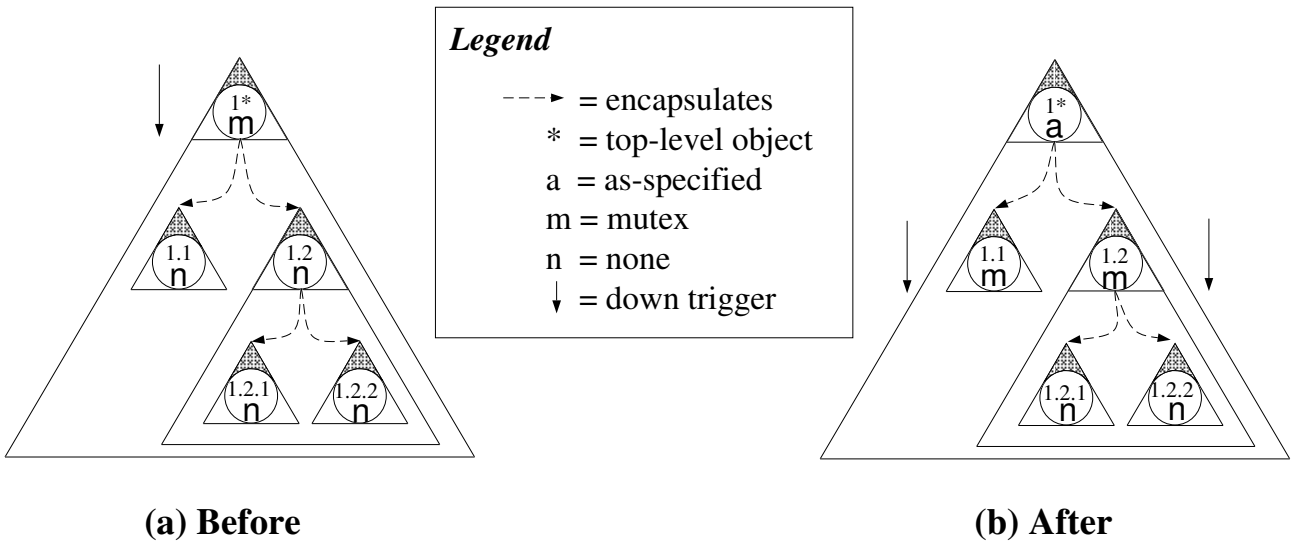

Figure 6: "Before" and "after" states of an encapsulation hierarchy that has moved its concurrency controllers down.

through the introduction of *restricted concurrency*. Restricted concurrency only allows transactional threads that are ancestors or descendants of the active transaction to proceed. No *new* transactions are allowed to enter the transition hierarchy.

The reason why there can be no transactional threads active in the encapsulated hierarchy is because there is, not enough information in the "before" hierarchy to construct the "after" hierarchy (See Figure 6). For example, it is not possible to convert a mutex object into an as-specified object while there is a transaction active in the mutex object. The problem is that a mutex object does not contain any concurrency control information for individual variables. Implicit as-specified objects (one concurrency controller per variable) need this information.

Hermes/ST objects encapsulated in a hierarchy are not necessarily confined to one node of a distributed system. Each object resides on one node, but an encapsulated hierarchy may be distributed over many nodes. Failures of nodes or communication links can occur at any time. The "before" and "after" hierarchies of Figure 6 both ensure serialisability of transactions (see Section 4.2). Thus, the algorithm must guarantee that if there is a failure during the transition, any partial states generated by the algorithm leave the hierarchy so structured that transactions visiting it will be serialised.

This motivates the need for the *insist* operation of Step 2. The insist operation either accomplishes its task on the entire target group, or if any part fails, the downward movement is terminated. In this case the target group is the children of the mutex object and the task is to change their concurrency controller type to mutex. If the insist operation fails then the hierarchy may be left with two mutex objects in sequence. As noted in Section 4.1, wasteful orderings such as these do not defy serialisability.

The algorithm may also fail to inform some descendants the new position of the mutex object (Step 4). This does not reintroduce the sneaking thread problem of Section 4.2. Any uninformed objects may unnecessarily synchronise invocations to mutex objects, but they will never fail to synchronise calls to a none object. This is because any uninformed none objects always consider the mutex object to be *higher* in the encapsulation tree than it actually is.

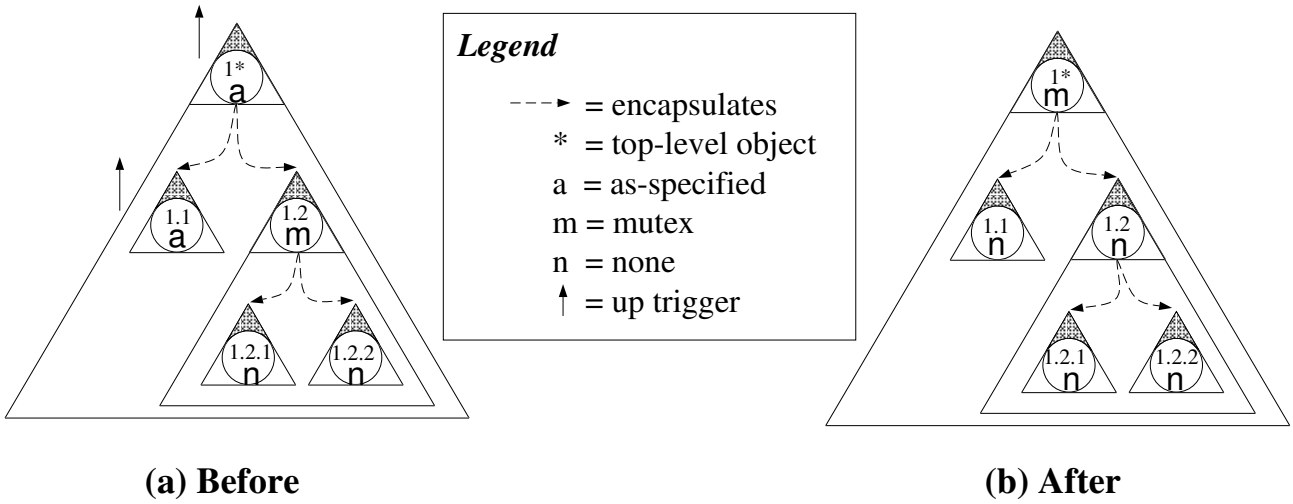

Figure 7: Before and after states of an encapsulation hierarchy that has moved its concurrency controllers up.

5.3 Upward Concurrency Controller Movement

Figure 7 (a) illustrates an example encapsulation hierarchy before it has moved its concurrency controllers up. The hierarchy consists of an as–specified object (O_1) and its associated up trigger, which encapsulates a sub-hierarchy of Hermes/ST objects ($O_{1.1}$, $O_{1.2}$, $O_{1.2.1}$, $O_{1.2.2}$). Figure 7 (b) shows the same example encapsulation hierarchy after it has moved its concurrency controllers up. The former as–specified object (O_1) has become a mutex object, and all encapsulated descendants ($O_{1.1}, O_{1.2}, O_{1.2.1}, O_{1.2.2}$) have become none objects.

Upward concurrency controller movement can be viewed as moving the mutex object up to the level of the up triggerring object. The algorithm to do this can be expressed in the following manner:

```
1. Introduce ‘‘restricted concurrency’’ to delay any new
   transactions attempting to enter the hierarchy.
2. Insist: [ all descendants do two things
 a) Inform the triggerring object of any
    asynchronous transactional threads commenced
    at this object. Also inform the triggering
    object when these transactional threads commit.
 b) Record the new mutex point. (Thus serialising
    any asynchronous transactional threads
    below the up triggerring object.)]
3. Wait until hierarchy contains only one transaction thread.
4. Inform all non-none descendants that they should
   become none objects.
5. Change the restricted concurrency object to a mutex object
   and re-schedule any delayed transactions from Step 1.
```

Unlike, downward concurrency controller movement it is not necessary to wait until *all* transactional threads (and hence transactions) active in the up triggerring object's hierarchy have committed. Upward

movement can be performed when there is *one* transactional thread active in the hierarchy. This is possible because an as-specified object has enough concurrency control information to convert to a mutex or a none object. There must only be one transactional thread active because a mutex (or none) object can only support one transactional thread at a time. Similarly, a mutex object has enough concurrency control information to convert to a none object.

In general, an encapsulation hierarchy containing as-specified objects can have more than one transaction active in it. An encapsulation hierarchy subsumed by a mutex object must only have one transaction active in it. Furthermore, the sneaking thread problem dictates that all transactional threads of the one active transaction must be serialised. Thus, in effect a mutex object's encapsulation hierarchy has only one transactional thread active at a time. Steps 1 and 2 of the algorithm are used to ensure the hierarchy is suitable for conversion to a mutual exclusion (mutex) hierarchy.

Step 1 changes the concurrency controller of the up triggerring object to support restricted mutual exclusion. As was the case with downward concurrency controller movement, restricted mutual exclusion only allows transactional threads that are ancestors or descendants of the active transaction to proceed. Again, no new transactions are allowed into the hierarchy.

Step 2a informs the up triggerring object about any transactional threads that have been generated by other as-specified or mutex objects in the hierarchy. It further informs the up triggerring object when these have completed. Step 2b ensures that no new transactional threads are generated in the up triggerring hierarchy (such asynchronous calls will be seen as sneaking threads and thus will be serialised.). Thus the information recorded and restrictions imposed by, steps 1 and 2, is sufficient to allow the up triggerring object to determine when there is only *one* transactional thread in its hierarchy.

As was the case for downward movement the "before" and "after" hierarchies of Figure 7 both ensure serialisability of transactions. The up algorithm must also guarantee that a network or communications failure during the transition from "before" to "after", will leave the encapsulation hierarchy so structured that serialisability is maintained.

If the insist operation fails then there will be some objects in the up triggerring object's hierarchy that are unnecessarily serialising asynchronous method invocations. As demonstrated in Section 4.2 this does not violate serialisability. Furthermore, there might be objects in the hierarchy that consider the mutex object to be at a *higher* position than it actually is. As was the case with the downward concurrency movement algorithm, this does not reintroduce the sneaking thread problem.

The algorithm may also fail to inform some descendants that they should become none objects (Step 4). This will potentially lead to an encapsulation hierarchy that contains as-specified or mutex objects below a mutex object. As noted in Section 4.1, these sorts of wasteful orderings do not defy serialisability.

6 Comparison with Other Systems

The author is unaware of any other systems that implement nested encapsulation and *enforce* it through access rules. Most object-oriented systems *encourage* encapsulation but there is no systematic prevention of its abuse. Languages such as Smalltalk [GR89], C++ [Str86] and Eiffel [Mey88] permit arbitrary deference once a reference (pointer) has been acquired. Thus nested encapsulation can be viewed as enforcing *information hiding* [Mey88, BL92] in a consistent, system wide, fashion.

The maintenance of the nested encapsulation hierarchy introduces a slight cost to object creation in Hermes/ST. Parent objects in the hierarchy need to update their containment structure for each new, non top-level, object created.

Many systems support some (but not all) of the various types of concurrency control present in Hermes/ST. Among these are Argus [Lis88], Avalon [EME91], Arjuna [Shr92] and Clouds [GRJL92]. The

specification of concurrency control in systems like Argus, Avalon and Arjuna is tied to the implementation structure. Thus an "atomic array of integers" has a different implementation to an "array of atomic integers". Static multi-granular concurrency control Hermes/ST allows the granularity of concurrency control to be specified *independently* to the object structure for the intended abstract data type. This is achieved by having highest and lowest markers as instance creation parameters.

Distributed Eiffel [GRJL92] supports large and fine grained objects. It has demonstrated that an application can be redeployed with different concurrency control granularities by re-calibrating which classes are to be instantiated as fine grained and which classes are to be instantiated as coarse grained objects. This corresponds to the Hermes/ST development strategy, but there is no support for instantiating multiple concurrency granularities from one class.

The author knows of no direct analogue to dynamic multi-granularity concurrency control, which allows the instantiation of a data-type to vary its concurrency granularity during execution. Intention locking comes closest and it and its relationship to dynamic multi-granularity concurrency control is discussed in the next two sections.

7 Intention Locking

7.1 Intention Locking in Database Systems

Intention locking [BHG87, GR93] has been introduced in order to allow transactions to individually determine the number of locks that they need to acquire in order to perform a task. Transactions that access many objects try to lock groups of objects at once, while transactions that access fewer objects try to lock them individually. Thus, intention locking trades off the high overhead of potentially high concurrency (many locks) with the lower overhead of less concurrency (few locks). *Lock escalation* [BHG87, GR93] allows transactions to reduce the number of locks they are acquiring *during* execution. Lock escalation can lead to deadlock, thus causing transactions to restart.

Intention locking was originally defined for non–nested transactions and necessitated the introduction of a logical *lock type graph* [BHG87]. The lock type graph is used to describe the structure over which intention locks will be acquired. For example, the *logical* lock type graph for a distributed database could be $Database - Area - File - Record$ [BHG87]. In addition, new locking modes called *intention write*, *intention read* and *shared intention write* were introduced and the lock *compatibility matrix* [BHG87] was extended to include them. Intention locks are less restrictive than their "real" counterparts. Thus, for example, two intention write locks do not conflict and can be held on the same resource by different transactions. Finally, an intention lock acquisition and release *protocol* [BHG87] was introduced to ensure that locks are acquired and released in such a manner that serialisability of transactions is ensured.

Using the $Database - Area - File - Record$ lock type graph introduced above, intention locking can be illustrated. A transaction that changes many Records in a File would acquire an intention write lock for the Database and one Area, whereas it would acquire a write lock on the File containing the Records to be altered. In contrast, a transaction that changed only a few records would hold intention write locks for the the Database, Area, and File, with individual Records being write locked as necessary. Thus the transaction changing many records will restrict concurrent access to the File but not have to acquire any Record locks, whereas the other transaction has to acquire locks for each Record changed but does allow other transactions concurrent access to other records in the file.

7.2 Intention Locking in Encina

Intention locking has been applied to distributed programming environments supporting nested transactions such as Encina's Trans-C [Lib92, Cor94]. As it is often not desirable to define a global lock type graph for a distributed system, individual lock type graphs are associated with each distributed component (server). As was the case for database systems, it is the responsibility of each function defined for a server to adhere to the server's logical lock type graph. The lock compatibility matrix, and lock acquisition and release protocol are the same as for intention locking in databases.

Nested transactions impose a further responsibility on the application coder. Any intention locks held by the parent transaction have to re-acquired by subtransactions. The reason for this is that if a transaction that owns a write lock over a group of objects allowed its subtransactions to automatically inherit this lock then non-serialisable transaction schedules would be possible. For example, a transaction that has acquired a write lock on a file should not spawn two asynchronous subtransactions that manipulate common records in the file. Each subtransaction must re-acquire its own write lock on the file to ensure serialisability. Thus, nested transactional calls represent another juncture where knowledge of intention locking is dispersed throughout a server. Lock escalation is not currently supported but could be implemented in the standard published manner [Lib92, Cor94].

7.3 Intention Locking in Hermes/ST

Hermes/ST also supports intention locking. In Hermes/ST, the logical lock type graph is defined automatically by the nested encapsulation hierarchy. When using intention locking it is again the responsibility of each method (function in Trans-C) to adhere to the encapsulating object's logical lock type graph. The lock acquisition and release protocol is the standard published protocol [BHG87]. The lock compatibility matrix is also standard [BHG87].

Hermes/ST approaches the problem of nested transactions and write locks over groups of objects in a different manner. Rather than insisting that the application developer re-acquire such write locks, and thus complicating a method's lock specification, Hermes/ST automatically serialises all such calls. These calls are treated in the same manner as sneaking threads (See Section 4.2). This usually decreases concurrency, but keeps the execution semantics consistent and minimises the spread of intention locking information throughout specifications. Lock escalation, although not currently implemented, could be performed as per the literature.

8 Multi-Granular Concurrency Control versus Intention Locking

8.1 Object Centered versus Transaction Centered Models

Intention locking, although supported, does have some deficiencies. Conceptually, intention locking (without lock escalation) is *static* in nature. The granularity of concurrency control has to be determined by the application developer when specifying the encapsulating object's class. Without changing this class specification or including more than one type of granularity in a specification, a method cannot react to execution time access patterns. Thus each method is encouraged to decide on the level of granularity that seems appropriate for it during specification. This is especially unattractive in a distributed system where client applications can have diverse and generally unpredictable access patterns that are not usually known at the time of specification of the server.

Lock escalation can be viewed as an attempt to work around the static nature of intention lock specifications. A transaction acquires locks according to the static specification until a *lock escalation*

threshold [BHG87] is reached. At this point the transaction considers the specification to be inadequate (e.g. too many locks acquired) and tries to reduce the number of locks it possesses through *lock conversion* [BHG87]. Analytic and simulation models provide the implied general "rules of thumb" about the desired granularity of locks. One such rule of thumb is that to maximise throughput longer transactions should acquire coarse granularity locks while shorter transactions should acquire finer granularity locks [BHG87, DA91].

The problem with the lock escalation approach to dynamically determining the granularity of concurrency control is that it encourages a *transaction centered* view of a distributed system. That a transaction is best placed to make such the tradeoff is debatable. For example, even though a transaction may write many records in a file, thus reaching its lock threshold, escalating this transaction's record locks to a file lock may be inappropriate. This could be the case if there are many other transactions accessing disjoint sets of records in the files. That some form of lock escalation is desirable is not being challenged. What is being challenged is whether or not a transaction, based on some local knowledge (such as a lock escalation threshold) should define its own concurrency control granularity. The encapsulating object is probably better placed to determine the appropriate level of locking granularity. This is because the object is aware of "the bigger picture" (e.g all transactions that are or have visisted it). The idea that an *object* owns resources and controls access to these resources is a key idea of object-oriented systems [Mey88].

Hermes/ST's dynamic multi-granular concurrency control supports such an *object centered* view of the granularity of concurrency control in an object hierarchy. An encapsulating object, based on the access patterns of *many* transactions can determine the appropriate granularity of its own concurrency controllers. In Hermes/ST, changing access patterns are detected by triggers that if necessary, change the potential parallelism through concurrency controller movement. Naturally, individual transactions can have input to the triggerring mechanism.

8.2 Specificational Differences

From a specificational viewpoint the need to maintain adherence to a *user–defined* lock type graph when using intention locking is also problematic in distributed object-oriented systems. Every new method that is added to an object has to explicitly acquire locks so as adhere to the lock type graph. Knowledge of the structure of this extraneous graph is therefore spread throughout each class and any consequent subclasses. As mentioned earlier, systems such as Encina's Trans-C computational model further exasperate this problem through the asynchrony introduced by nesting transactions.

Conversely, nested encapsulation provides Hermes/ST with an implicit lock type tree, viz the encapsulation hierarchy. Hermes/ST static multi-granular concurrency control localises knowledge about this hierarchy to just two instance creation parameters, the highest and lowest markers. Dynamic multi-granular concurrency control uses the *implicit* lock type graph to perform concurrency controller movement. Thus, if the addition of a new method does not affect the concurrency control granularity then nothing new need be specified. If concurrency control granularity is affected then only the markers and possibly some triggers have to be modified. As noted in Section 3, this allows developmental and reuse advantages.

8.3 Operational Differences

Despite its shortcomings, intention locking is still useful and therefore is supported in Hermes/ST. The first operational difference between intention locking and multi-granular concurrency control is a conceptual one. Intention locking allows individual methods to possess different concurrency control granularities on the same object structure at the same time. Conversely, multi-granular concurrency control forces

each method to share a common (although dynamic) model of this granularity.

An operational difference is that lock escalation under intention locking can lead to deadlock, whereas upward movement of concurrency controllers can not. This is because upward concurrency controller movement is delayed until there is only one transactional thread in the up triggerring object's encapsulation hierarchy. Lock escalation can lead to deadlock when two or more transactions try to escalate intention write locks to write locks on the same logical resource.

Another operational difference is that there is not a mechanism analogous to downward concurrency controller movement in intention locking. The need to sometimes increase potential concurrency during an encapsulated objects existence would appear to be useful. Intention lock specifications that are found to lock at too coarse a level have to be re-specified. As mentioned in Section 8.1, the static nature of these specifications is especially problematic in distributed systems.

The last difference is that upward concurrency controller movement can be more expensive than lock escalation. This is because all objects below the up triggering object in the encapsulation hierarchy have to be communicated with during concurrency controller movement[9]. Lock escalation, on the other hand, need only convert affected locks. Typically, there will be more objects in an encapsulation hierarchy than locks that need to be converted.

8.4 Co-Existence

Thus intention locking and multi-granular concurrency control co-exist. When *one* consistent object centered model of concurrency control granularity is required over *one* object topology then multi-granular concurrency control is easy to specify, develop, reuse and modify during application execution. When *more than one* model of concurrency control granularity is to be maintained over *one* object topology, then intention locking is probably better suited.

Intention locking and multi-granular concurrency control can also be used over an encapsulation hierarchy *at the same time*. Thus, when an encapsulation hierarchy moves its mutex objects towards the top, intention locking specifications are ignored. This still provides serialisable behaviour because mutex protected encapsulation hierarchies allow only one transactional thread at a time and therefore are more restrictive than the intention locking specifications that they override. This allows an application to provide a transaction centered concurrency control model that can be overruled by an object centered concurrency control model as is necessary.

9 Conclusions

This paper has introduced static and dynamic multi-granular concurrency control for the Hermes/ST distributed programming environment. It has also demonstrated that concurrency control granularity can be separated from individual abstract class descriptions. This allows individual classes to be instantiated with different concurrency control granularities. It furthermore allows these instantiations to vary their concurrency control granularities during program execution. Algorithms to do this were presented. Serialisability of nested transactions is maintained over the resultant object topology through the combination of nested encapsulation with restricted access rules, the order rule, and avoidance of the sneaking thread problem.

Hermes/ST, as presented in this paper, is an extension to [FHR93b]. The code is implemented in Smalltalk and executes over a cluster of SUN work stations.

[9] They must be informed of the new mutex object. See Section 5.3.

I wish to acknowledge the assistance of my colleagues Bernhard G. Humm and R. David Ranson in this research.

References

[Atk91] Colin Atkinson. *Object-Oriented Reuse, Concurrency and Distribution – An ADA-based approach.* ACM Press, New York, 1991.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[BL92] Gordon S. Blair and Rodger Lea. The impact of distribution on support for object-oriented software development. *Software Engineering Journal*, pages 130–138, March 1992.

[Cor94] Graeme Dixon Transarc Corporation. Personal communication, February 1994.

[DA91] S. Dandamudi and Siu-Lun Au. Locking granularity in multiprocessor databases. In *Procceedings of the 7th International Conference on Data Engineering*, pages 268–277. IEEE, 1991.

[EME91] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector (Eds.). *Camelot and Avalon.* Morgan Kaufmann Publishers, Inc., San Mateo, CA 94403, 1991.

[FHR93a] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Concurrency control for distributed nested transactions in Hermes/ST. In *Proceedings of the 1993 International Conference on Parallel and Distributed Systems (ICPADS'93)*, National Taiwan University, Taipei, Taiwan, Repubic of China, December 15-17 1993.

[FHR93b] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Hermes/ST user manual and technical manual. Technical Report No. 4, Telecommunications Software Research Centre, Department of Computer Science, University of Wollongong, Wollongong NSW 2500, Australia, 1993.

[FHR94] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Object-oriented extendibility in Hermes/ST, a transactional distributed programming environment. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, Lecture Notes in Computer Science 791 (to appear)*, pages 245–266. Springer-Verlag, 1994.

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

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, USA, 1993. ISBN 1-55860-190-2.

[GRJL92] L. Gunaseelan and Jr Richard J. LeBlanc. Distributed eiffel: A language for programming multi-granular objects. In *4th International Conference on Computer Languages*, San Franciso, California, April 1992. Available by anonymous ftp at IP address 130.207.5.20.

[Hum93] Bernhard G. Humm. An extended scheduling mechanism for nested transactions. In Luis-Felipe Cabrera and Norman Hutchinson, editors, *Proceedings of the Third International Workshop on Object-Orientation in Operating Systems (IWOOOS'93)*, pages 125–134, Ashville,

North Carolina, USA, December 1993. IEEE, IEEE Computer Society Press, Los Alamitos, CA.

[Lib92] IBM Hypertext Information Based Library. Aix dce base services 6000. CD-ROM, December 1992.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Cambridge, Great Britain, 1988.

[Mos85] J. Eliot B. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing*. MIT Series in Information Systems. The MIT Press, Cambridge, Massachusetts and London, England, 1985.

[Shr92] Santosh K. Shrivastava. The Arjuna system programmer's guide. Technical Report Public Release 1.0, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, February 1992.

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