A Kernelized Architecture for Multilevel Secure Object-Oriented Databases Supporting Write-Up

Roshan K. Thomas and Ravi S. Sandhu[1]

Center for Secure Information Systems &
Department of Information and Software Systems Engineering
George Mason University, Fairfax, Virginia 22030

December 16, 1993

[1]The work of both authors was partially supported by the National Security Agency through contract MDA904-92-C-5140. We are grateful to Pete Sell, Howard Stainer, and Mike Ware for making this work possible.

Abstract

This paper presents a kernelized architecture (i.e., an architecture in which no subject is exempted from the simple-security and $\star$-properties) for multilevel secure (mls) object-oriented database management systems (DBMS's) which support write-up. Relational mls DBMS's typically do not allow write-up, due to integrity problems arising from the blind nature of write-up operations in these systems. In object-oriented DBMS's, on the other hand, sending messages upwards in the security lattice does not present an integrity problem because such messages will be processed by appropriate methods in the destination object. However, supporting write-up operations in object-oriented systems is complicated by the fact that such operations are no longer primitive; but can be arbitrarily complex and therefore can take arbitrary amounts of processing time. We focus on support for remote procedure call (RPC) based write-up operations. Dealing with the timing of such write-up operations consequently has broad implications on confidentiality (due to the possibility of signaling channels), integrity, and performance.

We present an asynchronous computational model for mls object-oriented databases, which achieves the conflicting goals of confidentiality, integrity, and efficiency (performance). This requires concurrent computations to be generated within a user session, and for them to be scheduled so the net effect is logically that of a sequential (RPC-based) computation. Our work utilizes an underlying message filter security model to enforce mandatory confidentiality. We demonstrate how our computational model can be implemented within the framework of a kernelized architecture. In doing so, we present various intra-session and inter-session concurrency schemes. The intra-session schemes are concerned with the scheduling and management of concurrent computations generated within a user session, and we present conservative as well as aggressive scheduling algorithms. The inter-session schemes provide the traditional concurrency control functions of managing shared access to database objects, across user sessions.

Keywords: signaling channels, confidentiality, integrity, write-up, object-oriented databases, intra-session, inter-session, checkin/checkout, security kernel.

1 Introduction

The object-oriented paradigm continues to emerge as a useful and unifying one in computer science. It has borrowed ideas from such diverse fields as software engineering, artificial intelligence, and databases, and in turn advanced these fields in new directions. In light of this, we have seen several research and development efforts in object-oriented databases. The impetus for these developments can be attributed to emerging applications and computing environments that demand capabilities which are beyond those provided by record-based data models and conventional database technologies. Such applications and environments include computer-aided design, office automation, and cooperative work. From a data modeling perspective, object-oriented models not only allow the representation of complex object structures, but further allow modeling of the behavior of entities in a domain through methods encapsulated in objects.

As the object-oriented field is still maturing, there exists no single and precise definition of an *object-oriented data model* as observed by Maier [16]. However, there is some agreement on the core concepts that such a data model should support. We assume a data model that supports the following notions:

- *Object:* An object is an instance of an abstract data type, and is thus a unit that encapsulates some chunk of private state with a public interface.
- *Object Identity:* The object identity (object-id) uniquely identifies an object, and is further distinct from the internal state of the object.
- *Encapsulation of behavior:* An object supports operations that are implemented by methods (pieces of code). The state of an object is not directly manipulable, but can only be accessed by invoking one of the abstract operations defined in its public interface.
- *Class/Type:* Every object belongs to a type that is determined by its *class* (a class is akin to an abstract data type definition). Objects with the same structure and behavior can be grouped together as belonging to a class, thus enabling the sharing of information.
- *Class Hierarchy:* The data model should support the ability to organize classes into a class hierarchy. Classes in a hierarchy share definitions and behavior through the mechanism of *inheritance*. Classes lower in the hierarchy inherit from higher super classes. The inheriting class (and any corresponding instantiated object) is considered to be more specialized than its superclasses.

Variations along several themes of the core ideas above have been proposed in the literature. These include selective inheritance, class-less objects, and class-less sharing mechanisms such as delegation, to name a few. Discussion on some of these issues can be found in [12, 23, 26]. Although the debate on object-oriented data models continues, we fortunately do not need to settle the many issues in order to deal with multilevel confidentiality. In fact, we take a minimalist view that the dynamics of the object-oriented paradigm can essentially be captured by encapsulation and message passing. Objects can be considered

to be autonomous entities taking part in a distributed computation. Objects communicate with each other through messages. Message passing between objects can be synchronous or asynchronous. The receipt of a message results in the invocation of a method in the recipient object, with possible update of its state and the sending of further messages. In synchronous message passing, the sender's method is suspended until it receives a reply from the receiver object. This parallels the semantics of remote procedure calls (RPC's) in distributed systems.

From the security standpoint, the object-oriented model has strong appeal. In particular, there seems to be less of a modeling (semantic) mismatch between real-world entities in the domain being modeled and their object counterparts in the object-oriented representation. This makes it easier to specify, interpret, and implement access control and security policies in terms of objects rather than primitive abstractions or representations.

Recently, we have seen several models and prototypes for addressing mandatory confidentiality in object-oriented databases [7, 8, 9, 14, 19, 20, 25]. A common characteristic of most of these proposals is that the security policy to be enforced is expressed as a set of properties/constraints. For example, in [19], six properties are identified. The first (called the hierarchy property) requires that the level of an object dominate that of its class. This is required to permit inheritance along the class hierarchy. In contrast to these models, the message filter model proposed in [7] is based on the view that the task of enforcing mandatory confidentiality essentially reduces to that of controlling and filtering the exchange of messages between objects. The security policy is thus captured in a filtering algorithm and enforced by a message filter component. The main advantage of the message filter model is the simplicity and conceptual elegance with which mandatory security policies can be stated and enforced. The work we present in this paper utilizes the message filter model as its foundation.

In designing multilevel secure database management systems, one has to consider the conflict that arises between confidentiality and integrity [17]. This is because the requirements to enforce integrity constraints often result in secrecy being compromised. Conversely, guaranteeing secrecy may require tolerating lower degrees of integrity. The above tension has led to most mls relational database systems prohibiting "write-up" operations. To see this, consider conventional databases (such as relational systems) where the effect of arbitrary blind write-up operations on integrity is unpredictable and uncontrollable. Thus in a multilevel relational system, there exists the potential for a low-level subject to obliterate higher-level data. There is considerable ground for optimism as we reexamine this issue within the object-oriented framework. If objects can communicate solely through messages, then the properties of encapsulation and information hiding will ensure that an object state is updated only in controllable ways. Methods invoked due to receipt of messages from lower level objects will now have precise semantics.

The feasibility of supporting write-up operations is complicated by the fact that such operations are no longer primitive (such as read and write), but can be arbitrarily complex and therefore can take arbitrary amounts of processing time. This has broad implications on confidentiality (due to the potential for signaling channels), integrity, and efficiency. In this paper we focus on write-up actions where the intended semantics is RPC-based. The central point that we wish to make in this paper is that abstract RPC-based write-

up operations can be supported in multi-level object-oriented databases while meeting the conflicting goals of confidentiality, integrity, and performance. Our main contribution is an asynchronous computational model coupled with a multiversioning scheme that achieves these goals, as well as an elaboration of how this computational model can be implemented under a kernelized architecture. The computational model calls for concurrent computations to be generated on behalf of a user session whenever messages are sent upwards in the security lattice. Multiversioning and scheduling schemes are used to ensure that such concurrent computations preserve the originally intended RPC semantics.

The kernel (as in an operating system) performs the lower-level functions. In a secure system, the *security kernel* implements the security mechanisms of the operating system. The successful application of the security kernel approach to building secure systems is based on the theory that only a small fraction of the total functions in an operating system are needed to enforce security, and that these functions can be isolated into a security kernel. We present a kernelized architectural framework for implementing the above computational model. As there exists no trusted subjects[1] in such an architecture, the assurance of mandatory confidentiality comes directly from the operating system. Further, the absence of trusted (multilevel) subjects necessitates that the concurrent computations generated by a user session be scheduled and coordinated in a distributed fashion, as no system component has a global snapshot of the various computations as they progress. Database integrity now requires that these concurrent computations under distributed coordination produce the equivalent effect as computations that are serviced sequentially. It should be noted that if write-up operations were not supported, the architecture would be straightforward, as no concurrency is involved.

We present algorithms and techniques to handle intra-session as well as inter-session concurrency. The intra-session schemes are concerned with the scheduling and execution of the computations generated within a user session. We present two scheduling algorithms that represent extreme points in a spectrum of conservative and aggressive strategies. We also develop a framework and a metric for the analysis of a family of scheduling algorithms, all of which preserve integrity but offer varying tradeoffs between complexity and performance. The inter-session schemes provide the classical database functions of concurrency control, and thus pertain to how database objects can be shared in a secure and correct manner, across multiple user sessions. We present an approach to concurrency control based on the checkin/checkout paradigm. Our main objective is to show how such an inter-session scheme can mesh with the intra-session schemes developed in this paper. A complete treatment of inter-session mls concurrency control is outside the scope of this paper.

The work reported in this paper advances many of the ideas presented earlier in the literature [21, 22, 24]. Initial investigations of architectural issues in [21] were followed by the study of secure (signaling channel-free) scheduling algorithms [22]. A conservative scheduling algorithm that required no trusted subjects for its implementation was presented in [24]. In this paper we give in addition an aggressive scheduling algorithm, followed by

[1]The term "trusted" is used often in the literature to convey one of two different notions of trust. In the first case, it conveys the fact that something is trusted to be correct. In the second case, we mean that some subject is exempted from mandatory confidentiality controls; in particular the simple-security and ?-properties in the Bell-LaPadula framework. It is the latter sense of trust that we refer to in this paper.

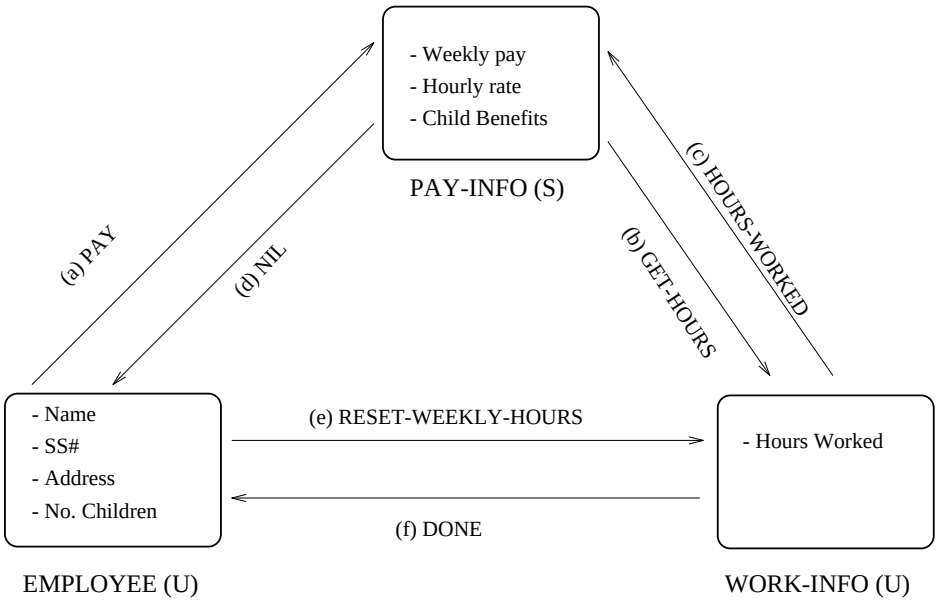

Figure 1: Objects in a payroll database

a framework for comparative analysis of different scheduling strategies, and finally various inter-session concurrency control schemes.

The rest of this paper is organized as follows. Section 2 motivates the issues addressed in this paper by means of an example. Section 3 gives an introduction to the message filter model and the filtering algorithm, and section 4 discusses a kernelized architecture. Section 5 presents the various intra-session scheduling schemes. This is followed by the inter-session concurrency control techniques in section 6. Section 7 concludes the paper and discusses avenues for future research.

2 A Motivating Example

We motivate the usefulness of write-up operations by an illustrative example. Consider a database for payroll applications, that has three objects: EMPLOYEE (Unclassified), WORK-INFO (Unclassified), and PAY-INFO (Secret), with the attributes shown in figure 1. In other words, every object is assigned a single level.[2] Weekly payroll processing is initiated by the lower level EMPLOYEE object with the sending of the (a) PAY message to the higher level PAY-INFO object. As the receiver is at a higher level than the sender, an innocuous NIL reply is returned by the message filter (as mandated by the message filtering algorithm, which will be discussed in the next section). On receiving the PAY message, the method in PAY-INFO sends a read-down message (b) GET-HOURS, to the lower level WORK-INFO object in order to retrieve the hours worked. This information is retrieved

[2] As explained in the next section, this does not pose any modeling limitations.

and returned in the reply message (c) HOURS-WORKED. However the method invoked in the lower level object WORK-INFO on receipt of the GET-HOURS message, is prevented (restricted) by the message filter from updating the state of object WORK-INFO. This is required to prevent write-down violations. Finally, the accumulated hours for the week is reset to zero by the message (e) RESET-WEEKLY-HOURS.

Another scenario for write-up arises when the child-benefits an employee is eligible for needs to be updated due to an increase in the number of children. Such an update is most efficiently accomplished by a trigger fired in the lower level EMPLOYEE object when the NO-CHILDREN attribute changes. The trigger would result in the sending of a message (with the value of number of children, NO-CHILDREN, as a parameter) to the higher level object PAY-INFO. The alternative to such a write-up would be that the PAY-INFO object scan the corresponding EMPLOYEE object for such changes, whenever the payroll is computed. However, this alternative imposes a significant performance cost for slow-changing information such as NO-CHILDREN.

The benefits of write-up operations in object-oriented databases come at the cost and complexity of implementation mechanisms needed to support them. The complexity arises due to the intrinsic abstract nature of object-oriented computations. Conventional databases generally have a flat view of data, and the operations are generally primitive reads and writes. Hence these operations may be assumed to take constant time.[3] Now contrast this with object-oriented systems where objects exhibit more complex structure and richer semantics. In this case, whenever a message is sent from a low level object to a higher one, we cannot assume that the invoked method in the receiver will terminate and return a reply in some constant time. Now, in the multilevel context, the actual reply from the higher level object cannot be returned to the lower level receiver as this will violate confidentiality. Hence we are faced with a fundamental dilemma. How and when do we resume a suspender sender method to mimic RPC semantics? Further, can this be done without violating confidentiality? In other words, a suspended sender method should be resumed in such a way that the sender cannot make any inference about processing at higher levels.

Since the actual reply from a high level receiver cannot be returned, let us assume for a moment that an innocuous reply such as a NIL is substituted and returned (as in the payroll example above). This assumption is made only for uniformity with the original message filter model [7] and for uniformity of coding. Thus we assume the receipt of the NIL reply as the logical point to resume a suspended sender method. If a reply (NIL or otherwise) is always guaranteed, the code for the sender method can be written to expect a reply whether the message is going up, down, at the same level or sideways in the lattice. Let us now examine how the timing of such a reply can have broad implications on confidentiality and integrity. To elaborate, consider the following alternate ways to deal with message replies:

- **Option 1:** Return a NIL reply on completion of the method in the receiver object;
- **Option 2:** Return the reply independent of the termination of the receiver method:

[3] In reality, even this is an approximation, albeit one that is normally made in the Bell-LaPadula style of models. Variations in read/write times occur due to caching, bus contention, disk buffers, demand paging, etc, and are usually manifested as timing covert channels.

- **Option 2a:** Return the NIL reply after some constant time interval that represents an upper bound for completion times;
- **Option 2b:** Return the reply after some random delay;
- **Option 2c:** Return the NIL reply instantaneously.

With the first option, we have a sequential execution of methods governed by remote procedure call semantics. However, the timing of a reply can now be modulated by the method in the higher level receiver object, and this opens up the potential for a signaling channel.[4] Thus under this first option, the integrity of the database is easy to maintain (as we have a simple sequential execution that requires no synchronization) but secrecy is compromised. The second category of options attempts to eliminate the above signaling channel by making it impossible for delivery of the NIL reply to be modulated by a higher level method. Option 2a imposes a heavy performance penalty whenever the receiver method has terminated and the sender remains unnecessarily suspended, waiting for the constant time interval to elapse. If we adopt option 2b, by randomizing the delay before returning the reply, we are faced with a tradeoff between performance and integrity. This is because if the reply is returned well after the termination of the receiver method, we are again unnecessarily holding up the sender method. On the other hand, if we return the reply too early (before the receiver method has terminated) we have to deal with the concurrent execution of methods. Concurrent executions introduce synchronization problems that can affect the integrity of the database. In particular, it is essential that the concurrent executions guarantee equivalence to a sequential (serial RPC-based) execution, as in the first option. When such equivalence can be guaranteed, we say that the concurrent computations preserve *serial correctness.* Note that this requirement of preserving serial correctness is entirely dictated by integrity considerations. From a confidentiality viewpoint, there is no need to synchronize these concurrent executions.

To illustrate a scenario of how the integrity of the database can be compromised, consider again the payroll database in figure 1. A sequential execution will lead to the message sequence *a, b, c, d, e, f;* while a concurrent execution may produce the sequence *a, d, e, f, b, c.* When weekly payroll processing is initiated by the sending of the PAY message from the lower level EMPLOYEE (U) object to the higher level PAY-INFO (S) object, a NIL reply is returned to object EMPLOYEE and the suspended method in EMPLOYEE resumes execution. Now it is possible for the RESET-WEEKLY-HOURS message (which resets the hours worked to zero) to be received and processed by object WORK-INFO before the message GET-HOURS. Thus the message GET-HOURS will retrieve the reset hours as opposed to the actual accumulated hours, resulting in an erroneous calculation of the weekly pay. In other words, with this second option, secrecy can be assured by eliminating the above category of signaling channels. However this is done at the cost of integrity.

Finally, option 2c above calls for replies to be returned instantaneously. We thus no

[4] In order to be precise, we distinguish between covert channels and signaling channels. A signaling channel is a means of downward information flow which is inherent in a data or computational model, and will therefore occur in *every* implementation of the model. A covert channel on the other hand is a property of a specific implementation, and not a property of the data/computational model. In other words, even if the data/computational model is free of downward signaling channels, an implementation may well contain covert channels due to implementation specific quirks.

longer incur the performance penalty that is possible with options 2a and 2b. However, we still have to address the integrity issue, as concurrent computations are now inevitable. We will demonstrate later in this paper how integrity can be achieved by the use of a multiversioning scheme that synchronizes concurrent actions on objects so as to guarantee serial correctness. To see how the multiversioning scheme applies to the payroll example, the (e) RESET-WEEKLY message would result in the creation of a new version of object WORK-INFO with the reset hours. However, an earlier version of object WORK-INFO that existed before the method in PAY-INFO was invoked, is used to process the (b) GET-HOURS message. Serial correctness is now ensured as the GET-HOURS message now retrieves the intended weekly accumulated hours as in the sequential execution.

The objective of satisfying the requirements of confidentiality, integrity, and efficiency within a kernelized architectural framework (i.e., without the use of trusted subjects) restricts us considerably in choosing one of the above options to deal with message replies. To start with, we observe that the signaling channels that arise with option 1 are only possible in an architecture with trusted subjects. Thus at first sight, it might appear that we could overcome this problem by utilizing a kernelized architecture. Unfortunately, option 1 is not implementable in a kernelized architecture as the $\star$-property prevents information flow from a higher level to a lower one, by disallowing write-downs. Such write-down operations are required to inform lower sender methods of the termination of higher level receivers. Option 2a and 2b are implementable in a kernelized architecture but at the cost of performance and integrity. Option 2c needs to address the integrity issue just as option 2b, but offers better performance than the latter (although as with option 2b, this comes at the cost of managing concurrency and multiversioning). Thus option 2c represents the best approach to handling write-up operations in a kernelized architecture. In summary we are forced to execute computations (methods) concurrently but nevertheless want to guarantee the original RPC-based semantics so as to preserve integrity (serial correctness).

3 The Message Filter Model

In this section we give some formal background to the message filter model. The model has evolved considerably since its original proposal. Our presentation in this section is limited to those aspects relevant to the understanding of this paper. For a more comprehensive discussion, the reader is referred to [7, 21, 22, 24].

3.1 The Message Filter Specification

Objects and messages constitute the main entities in the message filter model. As far as the security model is concerned, an entire object is classified at a single level. Modeling flexibility is not lost due to this as a user may model multilevel entities. The multilevel entities form a conceptual schema that is broken down into an implementation schema of single-level objects [7]. Messages are assumed, and required to be, the only means by which objects communicate and exchange information. Thus the core idea is that information flow be controlled by mediating the flow of messages. Consequently, even basic object activity such as access to internal attributes and object creation, are to be implemented by having

an object send messages to itself (we consider such messages to be primitive messages). The message filter takes appropriate action upon intercepting a message and examining the classifications of the sender and receiver of the message. It may let the message pass unaltered or interpose a NIL reply in place of the actual reply; or set the status of method invocations as restricted or unrestricted (explained later). The message filter is the analog of the reference monitor in traditional access-mediation models.

The message filter algorithm is given in figure 2. (In this and other algorithms, the % symbol is used to delimit comments.) Cases (1) through (4) deal with abstract messages, which are processed by methods. Cases (5) through (7) deal with primitive messages, which are directly processed by the security kernel. In case (1), the sender and receiver are at the same security level, and the message g_1 and its reply are allowed to pass. In case (2) the levels are incomparable and thus the filter blocks the message from getting to the receiver object, and further injects a NIL reply. Case (3) involves a receiver at a higher level than the sender. The message is allowed to pass but the filter discards the actual reply, and substitutes a NIL instead. (As we have argued, the timing of this NIL reply is a critical consideration.) In case (4), the receiver object is at a lower level than the sender and the filter allows both the message and the reply to pass unaltered.

In cases (1), (3), and (4) the method in the receiver object is invoked at a security level given by the variable *rlevel*. The intuitive significance of *rlevel* is that it keeps track of the least upper bound (lub) of all objects encountered in a chain of method invocations, going back to the root of the chain. The value of *rlevel* needs to be computed for each receiver method invocation. In cases (1) and (4) the *rlevel* of the receiver method is the same as the *rlevel* of the sender method. In case (3), *rlevel* is the least upper bound of the *rlevel* of the sender method, and the classification of the receiver object.

The purpose of *rlevel* is to implement the notion of restricted method invocations so as to prevent write-down violations. It is easy to see that if t_i is a method invocation in object o_i then *rlevel*$(t_i) \geq \mathrm{L}(o_i)$. We say that a method invocation t_i has a *restricted status* if *rlevel*$(t_i) > \mathrm{L}(o_i)$. When t_i is restricted, it can no longer update the state of the object o_i, it belongs to. We can visualize chains of method invocations as belonging to a tree such as in figure 3. Restricted method invocations in these chains now show up as restricted paths and subtrees. In figure 3, t_k represents a method in object o_k that sent a message, and t_n represents the method invoked in the receiver object o_n. The method t_n is given a restricted status as $L(o_n) < L(o_k)$. The children and descendants of t_n will continue to have a restricted status till such point as t_s. At t_s, the restricted status is removed since $L(o_s) \geq L(o_k)$ and a write by t_s to the state of o_s no longer constitutes a write-down violation.

The cases (1) through (4) that we have seen so far deal with abstract messages. However abstract messages will eventually lead to the invocation of primitive messages. These include **read**, **write** and **create** (cases (5) through (7)).[5] Now **read** operations always succeed, while **write**s succeed only if the status of the method invoking the operation is unrestricted. Thus if a message is sent to a receiver object at a lower level (as in case (4)), the resulting method invocation will always be restricted and the corresponding primitive **write** operation will not succeed. This will ensure that a write-down violation will not

[5]The delete operation has not been directly incorporated into the model. It can be viewed as a particularly drastic form of write and is subject to the same restrictions.

% *let* $g_1 = (h_1, (p_1, \ldots, p_k), r)$ *be the message sent from* o_1 *to* o_2 *where*

% h_1 *is the message name,* $p_1, \ldots, p_k$ *are message parameters,* r *is the return value*

if $o_1 \neq o_2 \vee h_1 \notin$ {**read, write, createg then case**
% *i.e.,* g_1 *is a non-primitive message*

(1) $L(o_1) = L(o_2)$: % *let* g_1 *pass, let reply pass*
invoke t_2 **with** $rlevel(t_2) \leftarrow rlevel(t_1)$;
$r \leftarrow$ reply from t_2; **return** r **to** t_1;

(2) $L(o_1) <> L(o_2)$: % *block* g_1*, inject NIL reply*
$r \leftarrow$ NIL; **return** r **to** t_1;

(3) $L(o_1) < L(o_2)$: % *let* g_1 *pass, inject NIL reply, ignore actual reply*
$r \leftarrow$ NIL; **return** r **to** t_1;
invoke t_2 **with** $rlevel(t_2) \leftarrow \text{lub}[L(o_2), rlevel(t_1)]$;
% *where lub denotes least upper bound*
discard reply from t_2;

(4) $L(o_1) > L(o_2)$: % *let* g_1 *pass, let reply pass*
invoke t_2 **with** $rlevel(t_2) \leftarrow rlevel(t_1)$;
$r \leftarrow$ reply from t_2; **return** r **to** t_1;

end case;

if $o_1 = o_2 \wedge h_1 \in$ {**read, write, createg then case**
% *i.e.,* g_1 *is a primitive message*

% *let* v_i *be the value that is to be bound to attribute* a_i

(5) $g_1 = (\textbf{read}, (a_j), r)$: % *allow unconditionally*
$r \leftarrow$ value of a_j; **return** r **to** t_1;

(6) $g_1 = (\textbf{write}, (a_j, v_j), r)$: % *allow if status of* t_1 *is unrestricted*
if $rlevel(t_1) = L(o_1)$
then [$a_j \leftarrow v_j$; $r \leftarrow$ SUCCESS]
else $r \leftarrow$ FAILURE;
return r **to** t_1;

(7) $g_1 = (\textbf{create}, (v_1, \ldots, v_k, S_j), r)$: % *allow if status of* t_1 *is unrestricted relative to* S_j
if $rlevel(t_1) \leq S_j$
then [CREATE i **with** values $v_1, \ldots, v_k$ **and** $L(i) \leftarrow S_j$; $r \leftarrow i$]
else $r \leftarrow$ FAILURE;
return r **to** t_1;

end case;

Figure 2: Message filtering algorithm

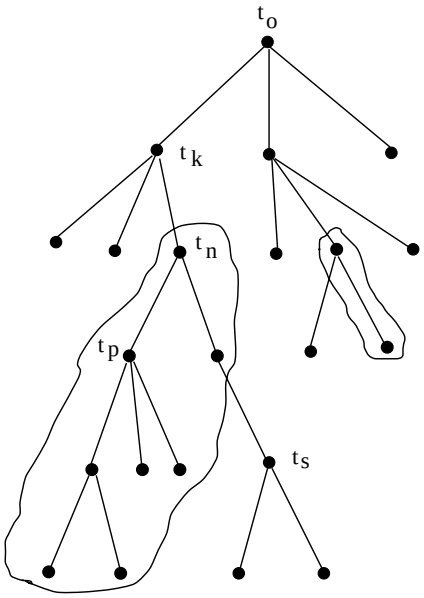

Figure 3: A tree with restricted subtrees

occur. Finally, the **create** operation allows the creation of a new object at or above the *rlevel* of the method invoking the **create**.

4 A Kernelized Architecture

Figure 4 illustrates our kernelized architecture. It is motivated and built upon the architecture of existing object-oriented database management systems. In particular, the demarcation into storage and object layers can be found in systems such as ORION, IRIS, and GEMSTONE [6, 11, 15]. The lower storage layer interfaces to the operating system and file system primitives, and is responsible for the management (i.e., the read, write, and creation) of typeless chunks of bytes representing objects. Every object is represented by a unique object-identifier. In our kernelized framework, the subset of this layer that is within the security perimeter consists of a single-level *storage manager* for every security level. A storage manager is responsible for the management of all objects at its level.

In contrast to the storage layer, the object layer is not typeless, but rather supports the abstraction of objects as encapsulated and typed units of information. This layer is thus responsible for implementing the object-oriented data model. It is important to note that much of the functionality required to implement the object-oriented data model lies outside of the scope of the TCB. Thus even support for the notion of objects as units of encapsulation, is provided by the object layer subset outside the TCB. Increasing the functionality of the object layer within the TCB, would increase its complexity, and would go against the design principles of security kernels.

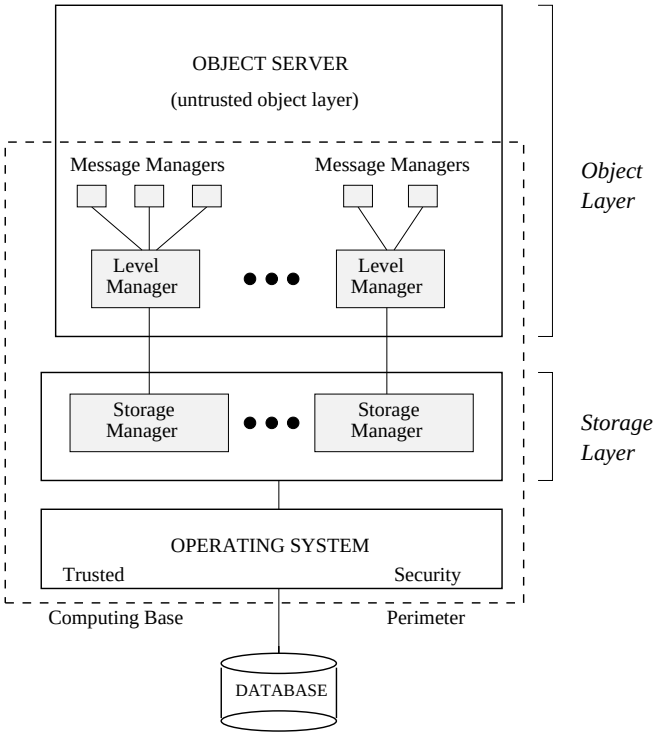

Figure 4: A kernelized architecture

The modules of the object layer that are within the security perimeter consists of level managers and message managers. A *level manager* is dynamically created for every level that can potentially have a method (computation) running[6], although conceptually we assume that it exists permanently to simplify our exposition. Its primary function is to coordinate the various computations (both queued and running) at a single level, and it is thus relatively long-lived. A *message manager* process is created dynamically whenever a message is sent upwards in the security lattice and concurrent execution of the sender and receiver methods is required. Once created, it implements the message filtering algorithm for the chain of methods emanating from such a concurrent receiver method. A message manager is thus a relatively short-lived process, and one that eventually terminates along with the last method in the associated chain.

The security perimeter of the object layer exports the following operations: **send, quit, read, write,** and **create**. The **read**, **write**, and **create** handle primitive messages. The system primitives **send** and **quit** are used by methods to send messages and replies. The interface between a message manager and a level manager consists of two calls: (1) **fork** issued by a message manager to request creation of a new message manager (at a higher level), and (2) **terminate** issued by a message manager to its local level manager (i.e., the

[6] In our further discussions, we use the terms message managers, computations, and methods, interchangeably. A message manager is merely a concurrent computation executing a chain of methods.

procedure send(g_1, o_1, o_2)

% *let* $g_1 = (h_1, (p_1, \ldots, p_k), r)$ *be the message sent from* o_1 *to* o_2

% *where* h_1 *is the message name,* $p_1, \ldots, p_k$ *are message parameters, and* r *is the return value*

% p *is the parameter set* $p_1, \ldots, p_k$ *and lmsgmgr is the level of the message manager* t_1

if $o_1 \neq o_2 \vee h_1 \notin$ {**read, write, createg then case** % *i.e.,* g_1 *is a non-primitive message*

(1) $L(o_1) = L(o_2)$: **push-stack**(p); $t_2 \leftarrow$ **select** method for o_2 **based on** h_1; **execute** t_2;

(2) $L(o_1) \sim L(o_2)$: **write-stack**(NIL); **resume**;

% *Let rstamps be a vector that is passed to a forked message manager*

(3) $L(o_1) < L(o_2)$: **append-rstamps-vector**(rstamps, wstamp);
fork($lmsgmgr$, lub[$lmsgmgr, L(o_2)$], fork-stamp, rstamps);
wstamp $\leftarrow$ wstamp + 1;
write-stack(NIL); **resume**;

(4) $L(o_1) > L(o_2)$: **push-stack**(p); $t_2 \leftarrow$ **select** method for o_2 **based on** h_1; **execute** t_2;

end case;

if $o_1 = o_2 \wedge h_1 \in$ {**read, write, createg then case** % *i.e.,* g_1 *is a primitive message*

(5) $h_1 =$ **read** : **if** $L(o_1) =$ lmsgmgr **then** v $\leftarrow$ wstamp **else** v $\leftarrow$ local-stamp ($L(o_1)$);
read o_1 with version $\leftarrow$ max{version: version $\leq$ v};

(6) $h_1 =$ **write** : **write** o_1 with version $\leftarrow$ wstamp;

% *Let o be the object-identifier of the new object created at level* S_j

(7) $h_1 =$ **create** : **create** o **with** $L(o) \leftarrow S_j$ **and** version $\leftarrow$wstamp; **write-stack**(o);

end case;

end procedure send;

procedure quit(r)

pop-stack;
if empty-stack then terminate(lmsgmgr, wstamp, fork-stamp)
else [**write-stack**(r); **resume**;]

end procedure quit;

Figure 5: Message manager algorithms for SEND and QUIT

level manager at the same level as the message manager) to terminate itself.

In reviewing the security perimeter of the above architecture, we wish to stress that the object layer plays no part in maintaining confidentiality. Now a primary objective in designing a kernelized architecture is to minimize the size of the security perimeter. Could we not then realize a secure database system without having the storage and object layers in the TCB (security perimeter)? If confidentiality were our only objective, the answer to the latter question would be "yes". The operating system alone would suffice to enforce the basic mandatory controls required to guarantee confidentiality. However, in a database system integrity is vital. This is why in our architecture, we have chosen to show the object and storage layers to be within the TCB. These modules are thus "trusted", but only in the sense being correct, and not in the sense of being exempt from mandatory controls. Even if correctness fails, these modules cannot compromise confidentiality by leaking information. In other words, the correctness of these modules can affect integrity but not confidentiality.[7]

The message filtering algorithm presented earlier can be thought of as an abstract (non-executable) specification of the filtering functions. An executable specification, as implemented by a message manager, is given in figure 5. As mentioned before, the **send** call is invoked by methods to send messages, while the **quit** is used to return replies. A stack is used to save the contexts associated with nested message sends. Whenever a message is sent by a method t_1 in an object o_1 to a second object o_2 at the same or lower level (cases (1) and (4)), the message manager saves the message parameters on a new stack frame, suspends execution of t_1, and begins execution of the method t_2 in object o_2. When t_2 terminates, the stack is popped and the return value from t_2 is recorded on the stack. The suspended sender method t_1 is then resumed, and it retrieves the return value from t_2 from the top frame of the stack. However, when messages are sent to incomparable or higher levels (cases (2) and (3)), a NIL value is recorded on the stack and t_1 is resumed immediately. In case (3) when a message is sent upwards in the security lattice, a message manager issues a **fork** call resulting in concurrent computations (as t_1 is resumed independently of the termination of t_2). The parameters of this call include the level of forking message manager, the level of the forked message manager, a unique fork stamp (identifying the start order in the equivalent sequential execution) for the forked message manager, and a vector (rstamps) of timestamps to process read down requests. Whenever a reply is returned and a message manager finds its stack to be empty, it means that there are no suspended methods waiting to be resumed. The message manager then issues a **terminate** call to its local level manager, to terminate itself.

The parameters of the terminate call include the level and fork stamp of the terminated message manager, as well as a timestamp identifying the last written version.

A message manager utilizes the following data structures in the algorithm.

[7] We believe it is misleading to assume that once a module is within the TCB, it will affect security. This is because security itself consists of three distinct, but inter-related areas: confidentiality, integrity, and availability. A module may be placed in the TCB for one or more of these reasons.

local-stamp:	a vector of timestamps to process read down requests, with one entry for each level dominated by the level of the message manager;
rstamps:	This is an incrementally constructed vector that is used to initialize the **local-stamp** structure.
fork-stamp:	a stamp identifying the message manager's fork order;
wstamp:	the write stamp for versions written by the message manager;

The **local-stamp** structure is needed to ensure serial correctness as it identifies the proper versions to read at all levels dominated by the message manager. This structure is initialized (partially) with the values of a vector **rstamps**, that is passed down by the ancestors of a message manager. When a message manager forks a new child computation, it appends the **rstamps** vector with a variable **wstamp**, and in turn passes it on to the new child (this is accomplished with a call to a predefined routine **append-rstamps-vector**, as shown in figure 5). The **wstamp** is a scalar variable which identifies the next version of objects that will be written/created at the level of the message manager. The **wstamp** is incremented by the message manager every time a **fork** request is issued. On the termination of the message manager, its **wstamp** is saved in the local level manager's **current-wstamp** variable. When a new message manager subsequently starts at this level, it will initialize its **wstamp** entry by reading off this **current-wstamp** value and incrementing it by one (as shown in algorithm **start** in figure 13). This increment is needed to avoid the latest created versions from being initially overwritten.

In moving from an abstract to an executable specification, we have so far described how the filter allows and blocks messages, and how return values are set to NIL. Now it remains to show how the notions of *rlevel* and restricted method invocations are implemented. The basic idea is very straightforward. Every message manager (process) is assigned a security level that is equivalent to the *rlevel* assigned in the filtering algorithm, and all methods executed by a message manager run at this level. The effect of restricted method invocations is now achieved by the enforcement of the standard $\star$ property in the Bell-LaPadula type security models [2]. In other words, whenever a method's status is restricted, its level (and the level of its message manager) will be higher than the object accessed, and the $\star$ property will prevent any write-down attempts.

As all our modules are single-level, our architecture needs to handle polyinstantiation of object identifiers. What if a subject requests the creation of a low-level object with an object-id that has already been assigned to a previously created higher level object? If the request is honored, we have an integrity problem as the object identifiers are no longer unique. If the request is rejected a signaling channel may be opened up. A solution would be to consider the user given identifiers as logical ones mapped to unique physical object identifiers that are system derived. The address space for the physical object identifiers could now be partitioned across the security levels. This will ensure that objects at different levels are not assigned the same physical id. An alternative to partitioning the physical address space would be to rely on a system component that generates identifiers in some cryptographically-strong pseudo-random fashion. This ensures that object identifiers cannot be used as a signaling channel.

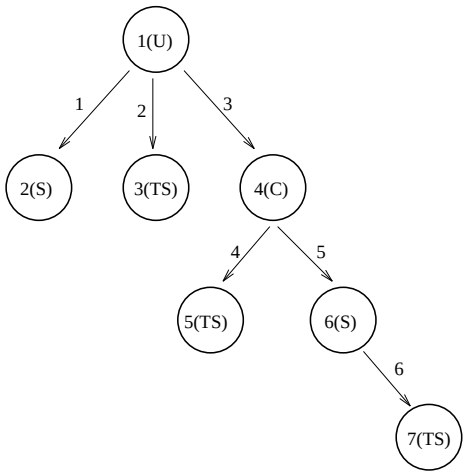

Figure 6: A session's computation tree of concurrent message managers

5 Intra-session Concurrency and Scheduling

In this section we focus on issues related to concurrency and scheduling within a single user session. We begin by discussing the notion of serial correctness and how this governs the degree of concurrency that can be allowed within a session. Maintenance of serial correctness requires that we capture the serial order of computations. This is done by means of a hierarchical scheme to generate forkstamps. Two extreme scheduling strategies both of which preserve serial correctness, but offer varying degrees of concurrency, are then discussed. Finally we present a framework for the analysis of these and other scheduling schemes.

5.1 Serial correctness versus concurrency

In section 2 we discussed the synchronization problem caused by concurrent computations and how this can affect serial correctness. To elaborate in more general terms, we can visualize a set of concurrent computations as a computation tree such as that shown in figure 6. In this figure we see that message manager 1 at the unclassified level has sent messages to one secret object, one top-secret object, and one confidential object in this sequence (we consider message manager 1 to be the ancestor (parent) of the three). As these objects are higher in level than unclassified, message filtering has resulted in the creation and concurrent execution of message managers 2, 3, and 4 as children of the root message manager 1.

We can now formally define serial correctness in terms of such a tree.[8]

[8]It is important to realize that even though the notions of serial correctness and serializability may appear to be analogous, they are not equivalent. Serializability theory in classical transaction management and concurrency control realms reasons about correctness and integrity in terms of the fundamental abstraction of a "transaction". Serial correctness on the other hand, is a more primitive notion as it does not recognize

Definition 1 *We say a session preserves serial correctness if for any computation* `c` *in the session's computation tree, and running at level* `l`*, the following hold:*

1. `c` *does not see any updates (by reading-down) of lower level computations that are to its right (in the tree);*
2. *For any of* `c's` *ancestor computations* `a`*, (i.e., any computation on the path from the root to* `c`*)* `c` *should see only the latest updates made by* `a` *just before* `a's` *child (or* `c` *itself) on this path was forked.*
3. *For any level* `k` *that is not the level of an ancestor of* `c`*, and* `k` $<$ `l`*,* `c` *should see the latest updates made by the rightmost terminated computations at level* `k` *that are still to the left of* `c`*.*

Given the above definition, let us see the complications concurrency poses to the maintenance of serial correctness. Now if we were to execute the above tree sequentially, the messages sent to higher level objects would be processed in the order given by the labels on the arrows. Note that this order can be derived by a depth-first traversal of the tree. However, with concurrent execution it is possible that message managers 4(C) and 6(S) may terminate well ahead of 3(TS). Therefore our synchronization schemes must ensure that message manager 3 does not see any updates by message managers 4 and 6, since 4 and 6 are to the right of 3.

Solving the above synchronization problem using classical techniques, such as those based on locking and semaphores, is known to be insecure as they open up signaling channels. Also, it is not possible to implement them in a kernelized architecture without introducing trusted subjects (since we need the ability to write-down and read-up). Our solution instead relies on a multiversioning scheme. The scheme calls for multiple versions of objects (created and accessed by a session) to be kept in memory. Such versions are invisible to other user sessions. Of course, if object versions are in virtual memory, they may migrate to the disk, but will still be unavailable to other sessions.[9] Each version is uniquely identified with a timestamp, and can be thought of as a checkpoint in the overall progress of a tree of computations. Thus although 4(C) and 6(S) may terminate well ahead of 3(TS), we are guaranteed that a read-down request from 3(TS) will always read versions that existed before 4(C) and 6(S) were started.

Given a computation, say c, the multiversioning scheme suggested above can provide synchronization when other computations to c's right (in the tree) get ahead. But to guarantee serial correctness, we must in addition ensure that c itself does not get ahead of earlier forked computations to its left. For example, under a sequential execution of the tree of

---

the abstraction or semantics of transactions, and is further more restrictive as it allows only a single serial order (ie., the order of an RPC-based serial execution of computations (methods)). However, if we were to map individual computations to transactions and derive the transaction serialization order from the forkstamps, serial correctness amounts to a stricter form of the multiversion concurrency control notion of one-copy serializability [3]. We intentionally do not give such a definition as this would give the impression that we are dealing with transactions, and would further introduce unnecessary formal machinery in our exposition.

[9] Inter-session object sharing and visibility is discussed in the next section on inter-session concurrency control.

computations in figure 6, we would expect message manager 2(S) and its descendants (if any) to terminate before message manager 3(TS) to its right, is started. Message manager 3(TS) should thus see all the latest updates by 2(S) and any of its descendants. Allowing arbitrary concurrency may not ensure this. Thus, in addition to multiversion synchronization, we need to enforce some discipline on these concurrent computations by scheduling them in a manner that guarantees serial correctness.

A scheduling strategy which guarantees serial correctness must take into account the following considerations.

- The scheduling strategy itself must be secure in that it should not introduce any signaling channels.
- The amount of unnecessary delay a computation experiences before it is started should be reduced.

The first condition above requires that a low-level computation never be delayed waiting for the termination of another one at a higher or incomparable level. If this were allowed, a potential for a signaling channel is again opened up. Fortunately, such channels cannot be introduced in a kernelized architecture and the confidentiality of the scheduling strategy thus comes for free. The second consideration admits a family of scheduling strategies offering varying degrees of performance. Some of these are discussed later in the next section.

In summary, the maintenance of serial correctness requires careful consideration on how computations are scheduled as well as on how versions are assigned to process read down requests. Collectively we have to guarantee the following constraints (as discussed in section 5.2, we assume that every computation is assigned a strictly increasing forkstamp that is consistent with the start order in a sequential execution):
Whenever a computation c is started at a level l,

- **Correctness-constraint 1:** There cannot exist any earlier forked computation (i.e. with a smaller forkstamp) at level l, that is pending execution;
- **Correctness-constraint 2:** All current non-ancestral as well as future executions of computations that have forkstamps smaller than that of c, would have to be at levels l or higher;
- **Correctness-constraint 3:** At each level below l, the object versions read by c would have to be the latest ones created by computations such as k, that have the largest forkstamp that is still less than the forkstamp of c. If k is an ancestor of c, then the latest version given to c is the one that was created by k just before c was forked.

We state formally as a theorem the sufficiency of these constraints.

Theorem 1 *Correctness constraints 1, 2, and 3 are sufficient to guarantee serial correctness of concurrent computations in a user session.*

Proof:
Constraints 1 and 2 ensure that when computation c at level l is started, there will be no more writes/updates forthcoming from earlier forked non-ancestral computations (the ancestral computations of c are those that are on the path from the root to c, in the computation tree) This guarantees that write operations by non-ancestral computations at levels l or below (and, therefore inductively across all levels) will occur in the same relative order as in a sequential execution. Write operations from ancestral computations may however be issued in an order different from the sequential execution. Such out of order writes can affect the values obtained by later read operations from higher level methods. However, constraint 3 ensures that read down operations under concurrent execution will obtain the same state as in a sequential execution. To see this, consider any computation such as c at a level l. In a sequential execution all non-ancestral computations at lower levels and with smaller forkstamps than c, would have terminated before c. Thus higher level reads by computations such as c would obtain the last written versions by such non-ancestral computations. The ancestors of c on the other hand would be suspended in a sequential execution, waiting for c and its future children to terminate. Thus read operations issued by c should see the versions written by the ancestors just before they were suspended. Constraint 3 requires this and prevents c from reading out of order writes (versions) of its ancestors.□

The multiversioning scheme requires a new object version to be created with every **fork** request issued (message sent upwards in the security lattice). In the worst case, what is the maximum number of versions at a level l that need to be concurrently retained by a user session, at any given time? This is equal to the number of subtrees rooted at the immediate children of any computation at level l, with one or more non-terminated computations. From an integrity standpoint, prematurely purging older versions may cause high level methods to fail, on issuing read requests.

5.2 Maintaining global serial order

We now discuss an implementation consideration for our scheduling schemes which has to do with maintaining knowledge of the equivalent global serial order in which computations are forked within a user session. In scheduling various computations, such knowledge is used to determine when a computation will be started. In an architectural framework without multilevel trusted subjects, no single system component has a global view (such as the tree in figure 6) of the entire set of computations as they progress. In coordinating various computations, an individual level manager has to determine where in the global serial **fork** order, the computations at its level belong. One could be tempted to pursue a solution requiring the value of a global real-time clock to be appended to every message manager (computation) as it is forked. However, computations are not always forked in the equivalent serial order and thus a solution based on a real-time clock will not always work.

In [24] we proposed a hierarchical scheme to generate fork-stamps that is independent of the scheduling strategy used. The fork-stamps so generated, reflect the equivalent serial order of execution of the computations. Figure 7 shows a tree of computations and the fork-stamps generated for it. Every message manager (except the root) is assigned a unique

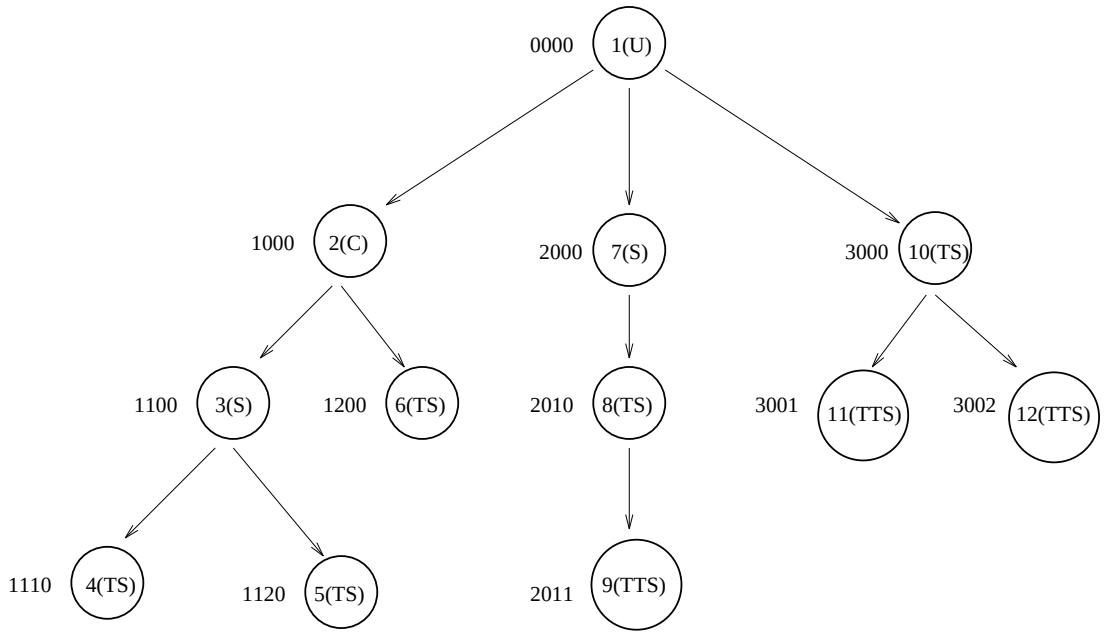

Figure 7: Generation of forkstamps for a session's computation tree

fork-stamp by the parent issuing the fork. The scheme starts by assigning an initial fork-stamp of 0000 to the root message manager 1(U). Every subsequent child of the root is then given a fork-stamp derived from this initial one by progressively incrementing the most significant (leftmost) digit by one. To generalize this scheme for the entire tree, we require that with increasing levels, a less significant digit be incremented. In general for a security lattice with a longest maximal chain of n elements, we need to reserve $p * (n - 1)$ digits for the forkstamp. In a lattice with l levels, and c compartments, $n = l + c$. The value of p would depend on the maximum degree of a node in a computation tree. For example if we assume that any computation sends a maximum of 99 messages to higher levels, then setting $p = 2$ would be sufficient.

5.3 A conservative level-by-level scheduling scheme

We now discuss a level-by-level scheduling scheme [24]. We characterize this approach as being conservative (as opposed to being aggressive) since the objective here is not to maximize concurrency. In other words, a computation may be unnecessarily delayed before being started even if its earlier execution would not violate serial correctness. Although this scheme may not always be optimal in terms of performance, it does give insights into how concurrent computations can be scheduled and completed in a simple, yet secure, correct, and distributed fashion. If in an application, the individual computations are of very short durations, the conservative scheme might be a good (or even the preferred) choice since it requires fewer data structures, is easier to implement, and the unnecessary delay induced,

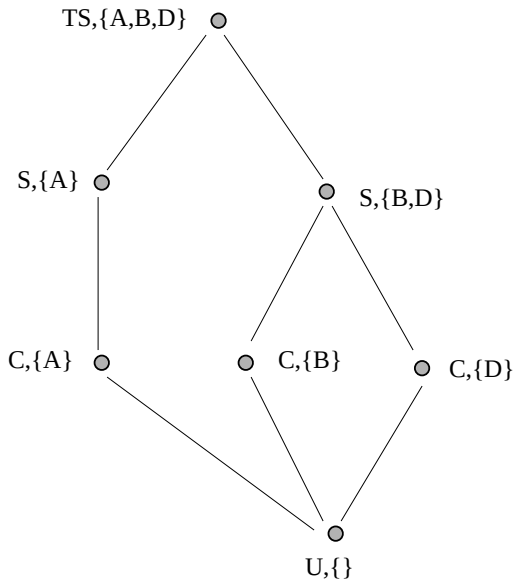

Figure 8: Conservative level-by-level scheduling in a lattice

tolerable.

The conservative scheme maintains the following invariant:

Inv-conservative: *A computation is executing at a level l only if all computations at lower levels, and all computations with smaller fork stamps at level l, have terminated.*

Thus the basic idea is to execute forked computations in a bottom up fashion in the lattice, starting with the lowest level. At any point, only computations at incomparable levels can be concurrently executing. We thus begin with the root computation and allow it to run to completion. Meanwhile, all higher level computations that are forked by the root are unconditionally queued (in forkstamp order) at these higher levels, by the respective level managers. Upon termination of the root, its level manager signals that it is okay to release computations at all immediate higher levels by sending a WAKE-UP message to these levels. Thus when a level manager receives a WAKE-UP message from *all* immediate lower levels, it proceeds to dequeue and execute computations at its level one at a time. Note that, at this point, this level manager is guaranteed that no more fork requests will be forthcoming from lower levels. Eventually, the level manager will find its queue to be empty. The next higher levels are then released through WAKE-UP messages.

For a more visual explanation of this level-by-level scheduling strategy, consider the lattice in figure 8. On termination of the root computation at level [U,{}], WAKE-UP messages are sent to all the immediate higher levels [C,{A}], [C,{B}], [C,{D}], and queued computations at these levels are then released. Next, computations at [S,{B,D}] are started when all those at the immediate lower levels [C,{B}] and [C,{D}] have terminated. Eventually,

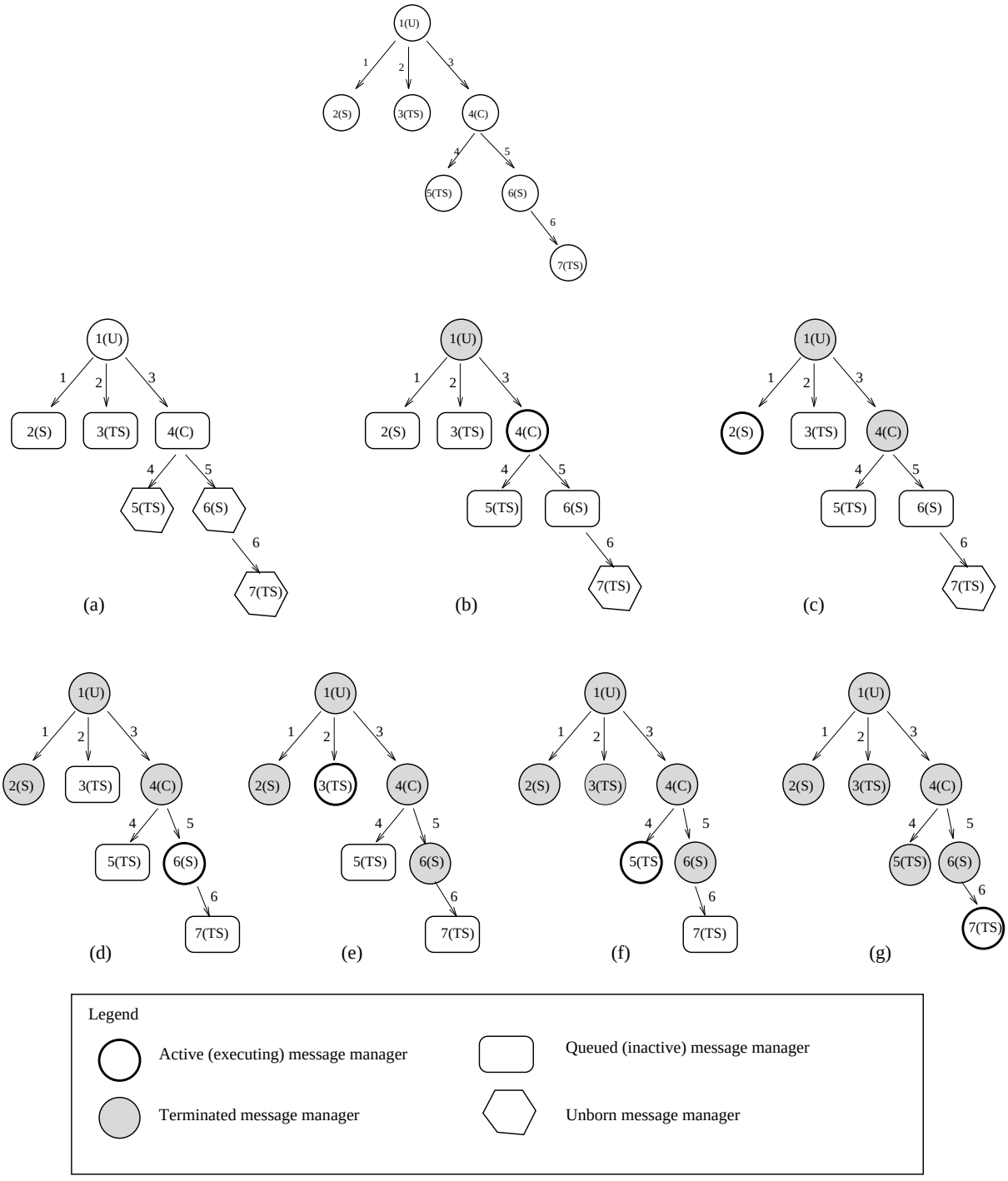

Figure 9: Progressive execution under conservative scheduling

computations at the highest level [TS,{A,B,D}] are started on the termination of computations at levels [S,{A}] and [S,{B,D}] followed by the receipt of a WAKE-UP message from each of these levels.

Figures 9(a) through 9(g) illustrate the progressive execution of the computation tree in figure 6, as governed by the level-by-level scheduling scheme. At each stage the termination of a computation results in the start-up of another. In this example, there can only be one computation executing at any given moment as the lattice is totally ordered. More generally, we could have multiple computations running, provided they are at incomparable levels. As shown in figure 9(a), the startup of the root computation has resulted in its forked children to be queued (the unborn computations have not yet been created, and are shown in the figures for visual completeness only). The subsequent termination of the root (see figure 9 (b)) has resulted in the forked child, at the lowest level 4(C), to be executed.

The level manager algorithms to implement this scheduling strategy are given in figures 10 through 13. The level manager data structures utilized in these algorithms are described below:

Level manager data structures:

current-wstamp:	the current timestamp given to objects written;
queue:	a queue of message managers waiting to be activated;
terminate-history:	a list of ordered pairs (fork-stamp, wstamp);

When a computation is forked, it is unconditionally queued by the local level manager, as shown in procedure **fork** in figure 10. When notified of the termination of a message manager at its level, a local manager dequeues and starts the next computation at the head of its local queue; if the queue is found to be empty, a WAKE-UP message is sent to all immediate higher levels (see procedure **terminate** in figure 11). When a level manager receives a WAKE-UP message from each of the immediate lower levels, it dequeues its local queue and starts the next computation; if the queue is empty, the WAKE-UP message is simply forwarded to all the immediate higher levels in the lattice.

A message manager's **local-stamp** vector is initialized in two phases, with the first one undertaken when a message manager is forked and the second one deferred until the message manager actually starts. For a message manager just forked, the first phase entries identify the versions to be read at the levels of ancestors, on the path from the root to itself (i.e., the path in a computation tree for a session). These first phase entries are actually obtained by a message manager from another vector that is passed along by its parent. Such a vector can be seen as one that is incrementally constructed along a path in the computation tree. To do this, every message manager is required to save the timestamps in the vector (rstamps) obtained from its parent and on issuing a **fork**, to reconstruct a new vector to give to its child (see figure 5). This newly constructed vector will contain the timestamps from the old vector appended with the write stamp **wstamp** at the level of the issuing message manager. Finally, in the second phase we obtain **local-stamp** entries for the levels that did not participate in phase one (this is done in the **start** procedure of figure 13). To enable this, every level manager maintains a **terminate-history** data structure that contains a list of terminated computations (identified by their forkstamps), and their associated **wstamp** values at termination time. At each level, the computation

```
Procedure fork(level-parent, level-create, fork-stamp, rstamps)
{
%Let level-create be the level of the local message manager
Create a new message manager mm at level level-create;

%Record the fork-stamp passed on by the parent
mm.fork-stamp ← fork-stamp

%Begin phase 1 of acquiring local-stamp entries
For (every level l ≤ level-parent)
do
      initialize mm.local-stamp table entries from rstamps;
End-For

%This is a priority queue maintained in fork-stamp order
enqueue(queue, mm);
}
end procedure fork;
```

Figure 10: Level manager algorithm for **fork** processing

```
Procedure terminate(lmsgmgr, wstamp, fork-stamp)
{
%Let tt be the message manager that just terminated at level lmsgmgr
%Let lm be the level manager at level lmsgmgr

%Update local current write stamp from tt
lm.current-wstamp ← wstamp

%Update local Terminate-history with the fork-stamp and wstamp of tt
Append-terminate-history(terminate-history, fork-stamp, wstamp);

If queue is not empty
then
      dequeue(queue, mm);
      start(mm);
Else
      Send a WAKE-UP message to all immediate higher level managers;
End-If
}
end procedure terminate;
```

Figure 11: Level manager algorithm for **terminate** processing

```
Procedure wake-up
{
%Proceed if the necessary condition has been met
If a WAKE-UP message has been received from all lower levels
then
      If the queue is not empty
      then
        dequeue(queue, mm);
        start(mm);
      else
        Send a WAKE-UP message to all immediate higher levels;
      End-If
End-If
}
end procedure wake-up;
```

Figure 12: Level manager algorithm for **wake-up** processing

```
Procedure start(nn)
{
%Let nn represent the message manager to be started
%Let lm represent the level manager managing nn

%Complete phase 2 of acquiring local-stamp entries
For (every level l lower than the level of nn for which no timestamp
has been obtained so far)
do
      nn.local-stamp[l] ← mm.wstamp;
      where mm is the message manager entry in the terminate-history at level l
      with max{fork-stamp: fork-stamp < nn.fork-stamp}
End-For

%Update the write stamp (wstamp) from the level manager
nn.wstamp ← lm.current-wstamp + 1;

%Begin execution of the message manager nn
execute(nn);
}
end procedure start;
```

Figure 13: Level manager algorithm for **start** processing

with the largest forkstamp that is still less than the forkstamp of the message manager to be started, is selected, and the associated version/timestamp is read into the corresponding **local-stamp** entry.

We conclude this subsection by giving proofs of correctness and termination for our conservative level-by-level scheduling algorithms.

Proof of correctness.

Theorem 2 *The conservative (level-by-level) scheduling algorithms maintain the invariant* **inv-conservative**.

Proof:
While there are two message manager algorithms, namely **send** and quit and four level manager algorithms **fork, start, terminate** and **wake-up**, we focus only the latter two for the proof. The **terminate** and the **wake-up** algorithms invoke the **start** procedure whereby computations get activated (started). It suffices therefore to show that these algorithms (procedures) maintain the invariant **inv-conservative**.

Consider the **terminate** procedure first. If we assume that the invariant holds (as a pre-condition) before the procedure was invoked, then it follows that there are no active or queued computations at level lmsgmgr or lower. Now if the start(mm) statement is reached, the following pre-conditions are true: (a) there exists one or more queued computations at level lmsgmgr; (b) the computation mm, with the lowest forkstamp will be started. The start(mm) statement further ensures the post-condition: (c) mm, being the computation with the smallest forkstamp is started, and there are no queued or active computations at lower levels. This implies (maintains) the invariant. On the other hand, if the start(mm) statement is not reached the invariant obviously continues to be true.

Consider the **wake-up** procedure next. From the **terminate** procedure we see that a WAKE-UP message is sent to all immediate higher levels only if there are no active or queued computations at or below the level that sent the message. Hence, when a WAKE-UP message has been received at a level say lwake, from all lower levels, the following are true:

d. there are no queued or active computations at levels lower than lwake;

e. there are no active or terminated computations at level lwake.

The latter condition is true since a computation can be started only as a result of a previous terminate at the same level or due to the receipt of a WAKE-UP message (and no terminate event would have occurred at lwake at this point). Thus when the statement start(mm) is executed, the following post-condition is true: (f) mm is the first computation to be activated at level lwake and there exists no queued or active computations at levels lwake or lower. This clearly maintains the desired invariant. Once again, if the start(mm) is not reached, the invariant continues to hold.□

Having shown how our algorithms maintain the invariant **inv-conservative**, we now argue how these algorithms preserve serial correctness by maintaining correctness constraints 1, 2, and 3. We state this below as a corollary.

Corollary 1 *The conservative (level-by-level) scheduling implementation under invariant* **inv-conservative** *maintains serial correctness.*

Proof:
When a computation c is started at a level l, the invariant **inv-conservative** requires all computations that are forked at level l with smaller forkstamps, to have terminated. This maintains correctness constraint 1. The invariant also requires that on the start of computation c, all computations at levels lower than l to have terminated. This requirement clearly maintains correctness constraint 2 since the constraint requires only computations with smaller forkstamps than c and at levels l or lower to have terminated. In other words, as far as lower level computations are concerned, the invariant **inv-conservative** is more restrictive than correctness constraint 2, and clearly maintains (implies) the latter. The **local-stamp** table entries collected in the first phase (at fork time) by c reflect versions identifying the states of objects written at the level of these ancestors before each successive child in the ancestral path was forked. The second phase entries on the other hand identify latest versions written at lower levels for which there were no ancestors. In summary, all read down operations that are mapped to the versions identified by the **local-stamp** entries, will read the same object states as in a sequential execution, and thus maintain correctness constraint 3.□

Proof of termination

In order to proceed with a proof of termination, we assume that once a method (computation) is started, it runs uninterrupted to completion. Obviously, such an assumption can be valid only if the body of the method contains no errors such as an infinite loop. We assume that there is some time-out mechanism in place, to handle such situations. We argue termination of individual computations (methods) by formally stating and proving the lemma below:

Lemma 1 *Once a computation is started, it is guaranteed to terminate.*

Proof: The proof follows from two observations:

1. Whenever a computation issues a **send** which results in a FORK, it is not blocked, but rather runs concurrently with the receiver computation. Thus, if a computation only issued forked new computations, it is guaranteed to run to completion and terminate (since only a finite number of FORK requests can be issued).

2. Whenever a method issues a **send** that does not result in a FORK, it will be blocked and in general this could result in a chain of blocked methods. However, there will always be a method executing and progressing to termination at the end of such a chain, and if are no cyclical **send** relationships, such a method will eventually resume its blocked predecessor. It follows that any blocked method will be resumed eventually and allowed to run to completion in finite steps.

We formally state as a theorem, that a session will eventually terminate.

Theorem 3 *Under the conservative scheduling scheme, all computations in a session will eventually terminate and thus guarantee the termination of a user session.*

Proof:
By induction on the number of security levels, n, at which computations are forked in a session.
Basis: Consider the basis with $n = 0$. Then the only level with active computations will not have any fork requests emanating from it. It follows from the second part of the proof of lemma 1 that the session is guaranteed to terminate.
Inductive Step: For the induction hypothesis assume that when n is equal to m, all computations terminate at the m levels and a WAKE-UP is sent to all immediate higher levels. For the inductive step consider $m + 1$ levels where level l_{m+1} is a maximal element in the security lattice and dominates a subset of the m levels. Now by the induction hypothesis, all computations at the m levels would have terminated and hence a WAKE-UP message would have been received at level l_{m+1} from all immediate lower levels *in* m. It now remains for us to show that a WAKE-UP is received at level l_{m+1} from all immediate lower levels (dominated by l_{m+1}) that never had active computations in the user session. These levels thus do not belong to m. The argument to show this can be made from the following: (1) The induction hypothesis guarantees that the root computations which are at the lowest level, say l_1, in m, would have terminated and sent a WAKE-UP message to all immediate higher levels; (2) WAKE-UP messages are always forwarded across empty levels. Hence all levels which dominate l_1 and in turn are dominated by l_{m+1} would have WAKE-UP messages forwarded through them. This guarantees that l_{m+1} would receive these messages from all immediate lower levels, and when this happens the computation at the head of the queue (which has the smallest forkstamp) will be dequeued and started. The termination of this first computation at level l_{m+1} leads to the startup of the next one in the queue. Every terminate results in the next computation in the queue to be subsequently started in turn. The queue will thus be progressively emptied in finite steps and all computations at level l_{m+1} would have then terminated. Thus the entire session will terminate. □

5.4 An aggressive scheduling scheme

We now describe an aggressive scheduling algorithm. It is governed by the following invariant:
Inv-aggressive: *A computation is executing at a level l only if all non-ancestor computations in the corresponding session with smaller fork stamps at levels l or lower, have terminated.*

We characterize this as an "aggressive" scheme as every attempt is made to execute a forked computation immediately. The above invariant implies that if a computation is denied immediate execution, then there must be at least one non-ancestral lower level computation with an earlier forkstamp, that has not terminated. The invariant ensures that the correctness constraints 1 and 2 are never violated. The correctness of read-down operations is again dependent on multi-versioning.

The implementation algorithms for the aggressive scheduling scheme are given in figures 14, 15, and 16 (the start algorithm is the same as in figure 13 for the conservative

```
Procedure fork-aggressive(level-parent, level-create, fork-stamp, rstamps)
{
%Let level-create be the level of the local level manager
Create a new message manager mm at level-create;

%Record the fork-stamp passed on by the parent
mm.fork-stamp ← fork-stamp

%Begin phase 1 of acquiring local-stamp entries
For (every level l ≤ level of the parent of mm)
do
      initialize mm.local-stamp table entries from rstamps;
End-For

%Check to see if a forked computation can be started immediately
If ∀ l ≤ level(mm), ¬∃ any computation c : (c.fork-stamp < mm.fork-stamp
      ∧ c ∉ terminate history at l)
      then
        start(mm);
      else
        %This is a priority queue maintained in fork-stamp order
        enqueue(mm);
end-if
}
end procedure fork-aggressive;
```

Figure 14: Processing **fork** requests under aggressive scheduling

```
Procedure wake-up-aggressive
{
dequeue(queue, mm);
start(mm);
}
end procedure wake-up-aggressive;
```

Figure 15: Processing **wake-up** requests under aggressive scheduling

```
Procedure terminate-aggressive(lmsgmgr, wstamp, fork-stamp)
{
%Let tt be the message manager that just terminated at level lmsgmgr
%Let lm be the level manager at level lmsgmgr

%Update local current write stamp from tt
lm.current-wstamp ← wstamp

%Update terminate history
Append-terminate-history(terminate-history, fork-stamp, wstamp)

%Check if a computation at level lmsgmgr can be started
If queue is not empty
then
      %Let mm be the computation at the head of the queue
      If ∀l < level(mm), ¬∃ c : (c.fork-stamp < mm.fork-stamp
      ∧ c ∉ terminate history at level l)
      then
        dequeue(queue, mm);
        start(mm);
      End-If
End-If

%Check if a computation at levels ≥ lmsgmgr can be started
For all levels l ≤ lmsgmgr
do
   If ∃ c ∈ fork-history at l with (level(c) > lmsgmgr ∧
   c.fork-stamp > tt.fork-stamp) : ¬∃ any computation k with (tt.fork-stamp < k.fork-
stamp
   < c.fork-stamp ∧ k ∉ terminate history at level(k) ∧ k is not an ancestor of c)
   % We checked to see if c was not preceded by a lower-level active or queued
   % non-parent computation in any of the fork-histories searched
   then
     Send a WAKE-UP message to the level manager of c at level ll;
   End-If
End-For
}
end procedure terminate-aggressive;
```

Figure 16: Processing **terminate** requests under aggressive scheduling

scheme). In addition to the data structures needed to implement the conservative scheme, the aggressive one requires that every level manager maintain a **fork-history** consisting of a list of ordered pairs (fork-stamp, level). This helps a level manager keep track of the fork requests generated at its level.

The major differences between the aggressive and conservative schemes as evident in these algorithms, can be summarized as follows:

- On being forked, a computation may be immediately started, if doing so would not violate the invariant **inv-aggressive** (see figure 14).
- The termination of a computation may result in the start-up of the next queued computation at the same level as well as multiple computations at other higher levels (as shown in figure 16).
- A wake-up is sent to a higher level only if there exists at least one queued computation pending execution at the higher level (see the second half of the algorithm in figure 16). The fork-histories at lower levels are examined to determine this.
- A level may receive multiple wake-up messages before all its queued computations are released.

We now elaborate on these algorithms. When a computation is forked (see the if statement in figure 14), we have to decide if it can be started immediately. A forked computation is started immediately if there exists no non-terminated computations at lower levels and with smaller forkstamps. We can determine all the computations forked at lower levels by examining the fork histories at these levels. We can further determine which of these computations have terminated by examining the terminate histories at these lower levels. When processing terminate requests, a similar check is made upon the termination of a computation at a level to see if the next computation (if any) at the head of the queue for this level, can be started (see figure 16). We also check to see if computations queued at higher levels can be released. We examine the fork histories at lower levels for computations that have been forked from these lower levels but have larger forkstamps than the just terminated computation (see the for statement in figure 16). Such computations with larger forkstamps can be started so long as they are not preceded by lower level non-terminated computations to the left (in the computation tree). A WAKE-UP message is sent to the level managers at the levels for which computations can be stared. On receiving such a message, a level manager dequeues and starts the next computation at the head of its queue (see figure 15).

Figure 17 illustrates how a tree of computations can advance to termination under the aggressive scheme. In particular, we note that the termination of a computation may result in multiple start-ups of others at higher levels (even with a totally ordered security lattice), so long as the invariant is not violated (see figure 17 (b) where computations 3 and 6 are started on termination of 2). We also observe that with aggressive scheduling, by the time the first three terminations have occurred, namely, 2(S), 3(TS), and 5(TS), the entire tree of computations has been released for execution (see figure 17(d)). Now compare the progress of this tree under conservative scheduling where the first three terminations as shown in figure 9 (d), still leaves three others queued and awaiting execution. In summary, the tree progresses to termination at a much faster rate, under the aggressive scheduling scheme.

We now give proofs of correctness and termination for the aggressive scheme.

Proof of correctness.

Theorem 4 *The aggressive scheduling algorithms maintain the invariant* **inv-aggressive**.

Proof:
We start with the **fork-aggressive** procedure in figure 14. We see that for the statement start(mm) to be executed, the following pre-conditions are true:

a. there exists no non-ancestral queued or active computations at or below level(mm) and with a smaller forkstamp than mm;

b. mm is the only computation at level(mm).

After computation mm has been started the condition (a) above still holds and thus the invariant is maintained. A similar argument can be made for the start(mm) statement in procedure **terminate-aggressive**. When mm is dequeued, condition (a) above holds, and since mm has the smallest forkstamp in the queue, the invariant is maintained after the execution of start(mm).

It now remains to show that the start-up of a computation due to the receipt of a WAKE-UP message at a level, will not violate the invariant. To see this, we observe that a WAKE-UP message is sent to a higher level (in the **terminate-aggressive** procedure) only if there exists a pending computation say, c (at the higher level) that was denied immediate execution at fork time. Further, c has to have the smallest forkstamp among others at its level and should not be preceded by active or queued (pending) computations at lower levels and with a smaller fork than itself. Thus on receiving a WAKE-UP message, a level meets all the necessary conditions to start a computation. The post-condition following the start(mm) statement in procedure **wake-up-aggressive** thus maintains the invariant.□

We now state and show how the invariant **inv-aggressive** maintains serial correctness under our implementation.

Corollary 2 *The aggressive scheduling implementation maintains serial correctness.*

Proof:
We basically have to show how the correctness constraints 1, 2, and 3 are maintained. For a computation to be dequeued and successfully started, invariant **inv-aggressive** requires all earlier forked computations at level l or lower, to have terminated. But this is what is precisely required to maintain correctness constraints 1 and 2. The argument for the maintenance correctness constraint 3 is independent of the scheduling algorithm used. Thus the earlier argument given for the conservative case still holds.□

Proof of termination

Theorem 5 *Under the aggressive scheduling scheme, all computations in a session will eventually terminate and thus guarantee the termination of a user session.*

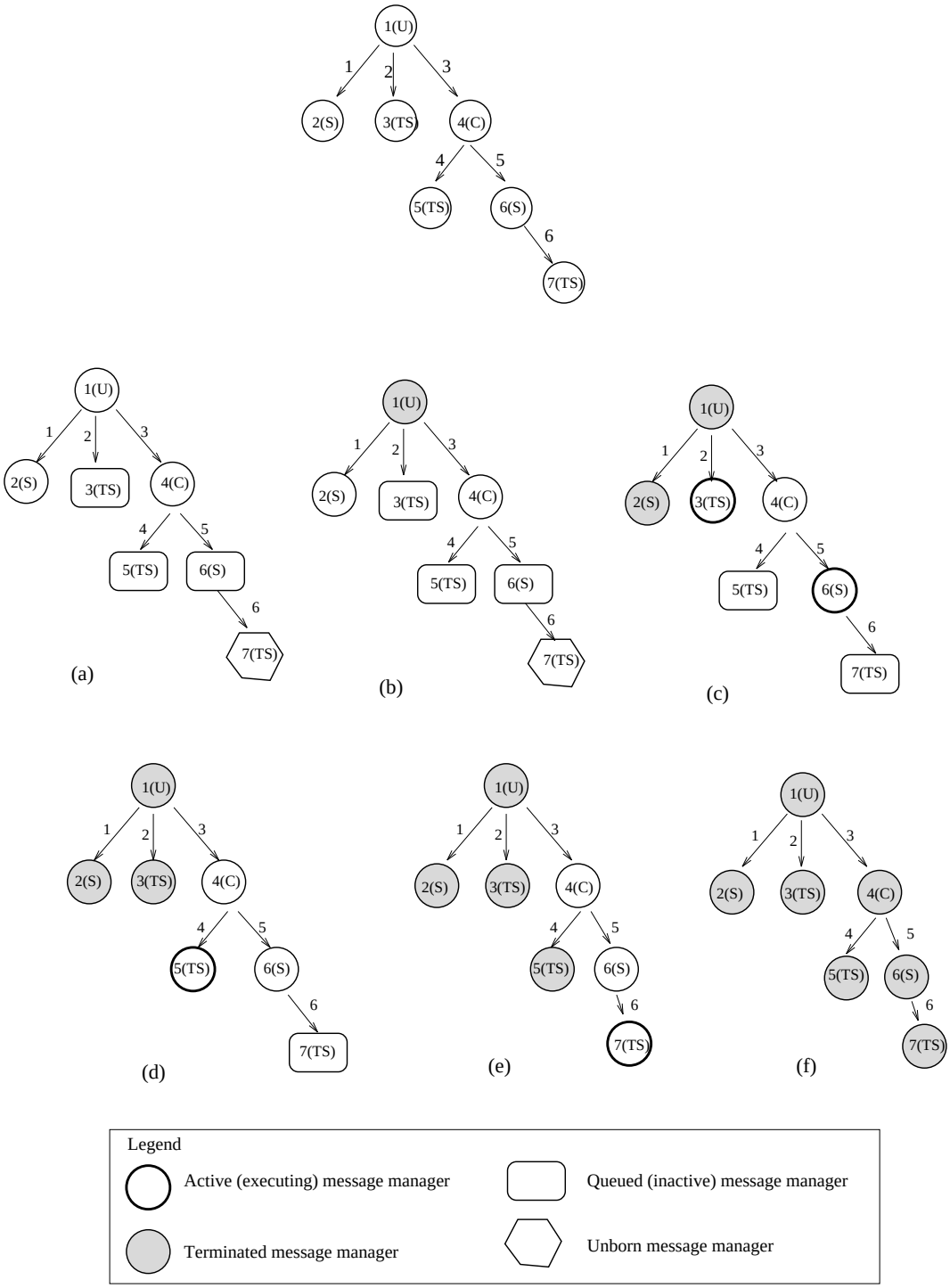

Figure 17: Progressive execution under aggressive scheduling

Proof:
To argue proof of termination for the aggressive algorithm, we observe that if a computation is denied immediate execution this can only be at fork time. Again we assume that once started, a computation is guaranteed to terminate (by lemma 1). Our task is thus basically to show that every queued/pending computation will eventually be started. Now if on fork, a computation f is denied immediate execution, then there must be at least one active computation say c, with a smaller stamp than f and at or below level(f). Now the termination of c is guaranteed by lemma 1. The termination of c will cause at least one computation with a greater forkstamp than c and a smaller forkstamp than f, or f itself, to be started. Now if f is not started, there can only be a finite number of computations that can potentially block f. Subsequent terminate events will progressively decrease the number of such computations with a smaller forkstamp than f. This will result in the eventual release of f for execution. With a similar argument, we can show that every queued computation will eventually be released for execution and thus run to termination. Thus the entire session will eventually terminate, concluding the proof. □

5.5 Analysis and Discussion

The conservative and aggressive schemes discussed above can be seen as two that approach the ends of a spectrum of secure and correct scheduling strategies. This is because it is meaningless to come up with any algorithm that does worse than the conservative one, in terms of the degree of concurrency allowed. At any given time, if there is a computation active at a maximal level in the lattice, then no other computations may be concurrently active. The conservative scheme thus exhibits the least (meaningful) degree of concurrency within a session. The only way to do worse would be to allow computations at incomparable levels in the lattice to execute one at a time. On the other hand with the aggressive scheme, we can potentially have concurrent computations running at every level. This can happen if a computation is forked at the highest level in the lattice, and this is followed by consecutive fork requests where each request is at the next lower level (and the lifetimes of these computations are long enough to overlap). One can always increase the degree of concurrency by exploiting intra-level concurrency. But conflicts at the same level can be easily handled by well-known concurrency control techniques. We do not explore this issue further in this paper as it lies outside the scope of the execution model and scheduling protocols we present.

In this subsection we briefly outline a framework for the comparative analysis of various scheduling schemes. In particular, we develop the notion of *delay-degree* as a metric for analyzing scheduling strategies. We demonstrate how by varying this metric, we can derive (and admit) a family of scheduling strategies offering varying degrees of concurrency, while guaranteeing confidentiality and serial correctness.

We begin with some definitions.

Definition 2 *A level is* inactive *if no computation is executing at the level.*

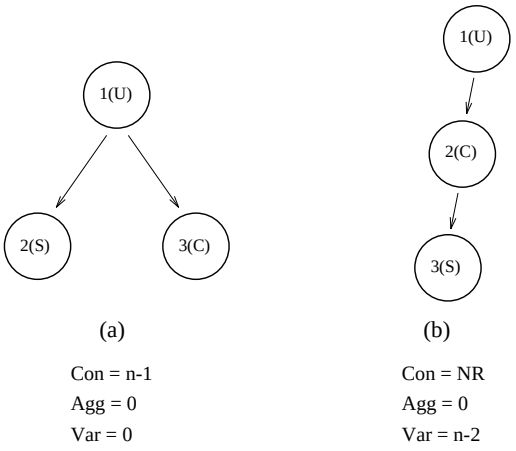

Figure 18: Full-enablers for a longest maximal chain of 3 elements (n = 3)

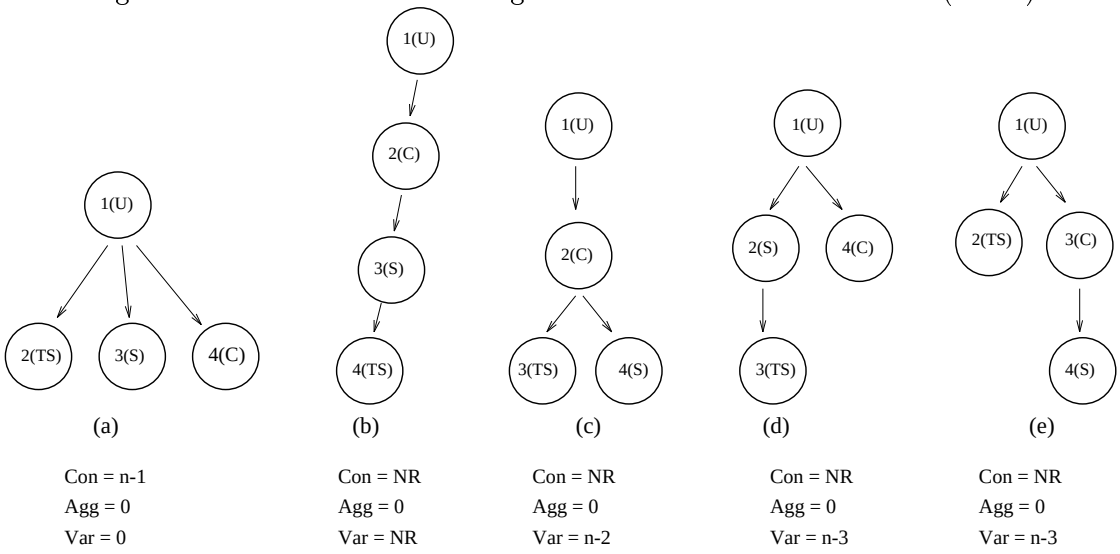

Figure 19: Full-enablers for a longest maximal chain of 4 elements (n = 4)

Definition 3 *A level is* active *if there exists an executing computation at the level.*

Definition 4 *We say a level l is* serial-execution enabled *(or* s-enabled *for short), if there exists at least one forked computation c, at l, and there are no active or queued non-parent computations with smaller fork stamps than c, at level l or below.*

Intuitively, when a level is s-enabled, executing the next computation at the head of the queue at this level will not violate serial correctness. A computation that is denied execution by a scheduling scheme when its level becomes s-enabled is therefore experiencing an unnecessary delay. We build on this observation and extend it below to an entire security lattice in order to formulate a metric for analysis purpose.

Definition 5 *A scheduling algorithm introduces an* unnecessary delay *whenever any level is s-enabled but remains inactive.*

Definition 6 *We say a chain of n security levels in a lattice is* fully-enabled *whenever every level in the chain is concurrently s-enabled.*

Definition 7 *We define a computation tree to be a* full-enabler *for a given security lattice, if it causes a longest maximal chain in the lattice to be fully-enabled.*

Thus when a maximal chain in the lattice is fully-enabled, computations can be concurrently running at every level in the chain. However, when scheduling is governed by some scheme, it is only certain scenarios that can lead to such chains being fully-enabled. We characterize below the computation trees associated with such scenarios as realizers.

Definition 8 *For a given scheduling algorithm and security lattice, we define a* realizable full-enabler *(or* realizer *for short) to be a full-enabler, which when scheduled by the algorithm, causes a longest maximal chain in the lattice to be fully-enabled.*

Definition 9 *We say a realizer has a* delay-degree (d-degree) *of k for some scheduling algorithm, if it causes k computations to experience unnecessary delays.*

Definition 10 *Given a security lattice (SC), a scheduling algorithm (A) is considered to have a* delay-degree (d-degree) *of k, where $k = \max\{$d-degree of all realizers for SC under $A\}$.*

Given a set of secure scheduling algorithms (schemes), we can now use their d-degrees as a basis for comparison. We thus need to derive the d-degree for any given scheduling scheme. To do this, we consider all the realizable full-enablers (realizers) and observe the maximum number of computations (excepting the root) that are denied immediate execution, on being forked. This number would give us the d-degree.

As an illustration, consider the full-enabler trees in figure 18 for a lattice with a longest maximal chain of three levels U, C, and S (where U < C < S). For the aggressive scheme, we see that both trees are realizers and in either cases no computation would be unnecessarily delayed. For the conservative scheme, only the tree in 18(a) is a realizer and we see that computations 2(S) and 3(C) would be unnecessarily delayed. For a further illustration, consider all the full-enabler trees for four levels U, C, S, and TS, as shown in figures 19(a) through 19(e). All the trees are realizers for the aggressive scheme, and in each case no computation would be unnecessarily delayed. However, only the tree in figure 19(a) is a realizer for the conservative scheme and the computations 2(TS), 3(S), and 4(C) would be unnecessarily delayed.

In both of the examples above, we see that the aggressive scheduling scheme would have a d-degree of zero (0), while the conservative scheme would have a d-degree of $n-1$ (for a lattice with a longest maximal chain of n elements). These results are general and not specific to these two examples. To be more precise, the d-degree, say k, of the conservative (or aggressive) scheme holds true for any lattice with a longest maximal chain of n elements, as long as $k \leq n$. Also, it follows that for any scheduling scheme with a d-degree of 0, a level is inactive only if it is not s-enabled.

Now are there other scheduling schemes that have d-degrees between the extreme values of 0 and $n-1$? To answer this question, we explore a variation of the level-by-level scheduling scheme. Recall that with the level-by-level scheme, computations are executed one level at a time. Thus at any given time, there is a *current-level* at which computations are dequeued and executed. While our variant would also require that computations be dequeued and executed one level at a time, it would in addition permit the execution of all the immediate child computations of any active computation at the current-level. To derive the d-degree of this variant, consider again the full-enabler trees in figures 18 and 19. Both trees in figures 18(a) and 18(b) are realizers with d-degrees of 0 and $n-2$ respectively, and thus giving a d-degree of $n-2$ for this variant (i.e., $\max\{0,\ n-2\}$). In figure 19 the trees (a), (c), (d), and (e) are realizers with d-degrees 0, $n-2$, $n-3$, and $n-3$ respectively, giving again a d-degree of $n-2$ for this variant. It thus introduces fewer delays (due to increased concurrency) than the conservative scheme with a d-degree of $n-1$. We conjecture that by varying the metric d-degree, one could derive several scheduling schemes.

6 Inter-session Concurrency Control and Object Sharing

So far we have looked at the issues of concurrency and scheduling within a single user session. We now focus on how objects can be shared across multiple concurrent user sessions. In the database literature, schemes to achieve this generally fall under the category of concurrency control and transaction management. Our purpose here is not to discuss a comprehensive

concurrency control scheme, but only to give a basic usable secure solution to object sharing across user sessions. Discussion of a comprehensive transaction model for multilevel systems is beyond the scope of this paper.

Our approach to object sharing is based on a checkin/checkout access data model [13]. There exists a public (single-version) database from which user sessions checkout objects as needed. The objects are checked out into local workspaces (private databases) of individual user sessions. When all activity associated with an object has ceased, the object is checked back into the public database. Due to concurrent activity in a user session, computations within a user session may view several versions of the same object. However, visibility across user sessions is limited to the public database which maintains only the latest version of every object.

6.1 Modeling user sessions as hierarchical transactions

In order to reason about the effects of concurrent user sessions on objects, we cast our solutions in terms of the familiar concept of transactions. For this, we present a model of a user session as a hierarchical set (tree) of multilevel subtransactions. We model the actions of each computation as a set of subtransactions. To be more precise, all the actions from the start-up to the issuing of the first fork request is considered to be one subtransaction. The subtransaction is considered to be running at the level of the corresponding computation. All actions between each subsequent pair of fork requests are considered to belong to individual subtransactions. Finally, all actions between the last fork request and the termination of the computation are modeled as one subtransaction. A subtransaction is considered to be the smallest unit of execution and is thus atomic. Thus if a subtransaction fails, it leaves the database objects unchanged, and as far as the database is concerned the subtransaction was never created. The atomicity property also means that operations from individual subtransactions cannot interfere with each other. In other words, subtransactions execute serially. Also, it follows from our hierarchical formulation above, that a subtransaction writes only at its level.

In figure 20, the computation 1(U) forks two computations 2(C) and 4(C) and is thus modeled as a set of subtransactions $1U_1$, $1U_2$, $1U_3$. All transactions generated by a single computation are numbered by a transaction stamp derived by concatenating the forkstamp of the computation with the next consecutive integer. This guarantees the uniqueness of the transaction stamps thus assigned. Modeling all the nodes in a tree results in a hierarchical (tree-like) structure of subtransactions. In figure 20, we see that there is a subtree rooted at $2C_1$. This means that a fork (of computation 3(S)) was issued by transaction $2C_1$. The subtree consists of all the transactions generated by the forked computation 3(S).

6.2 Multilevel checkin/checkout of objects

One of the considerations in designing an object sharing and transaction management scheme is that of formulating and maintaining some notion of inter-session correctness. Conventional database management schemes primarily support transactions that are short-lived and competitive. Interactions and visibility across such transactions are curtailed and

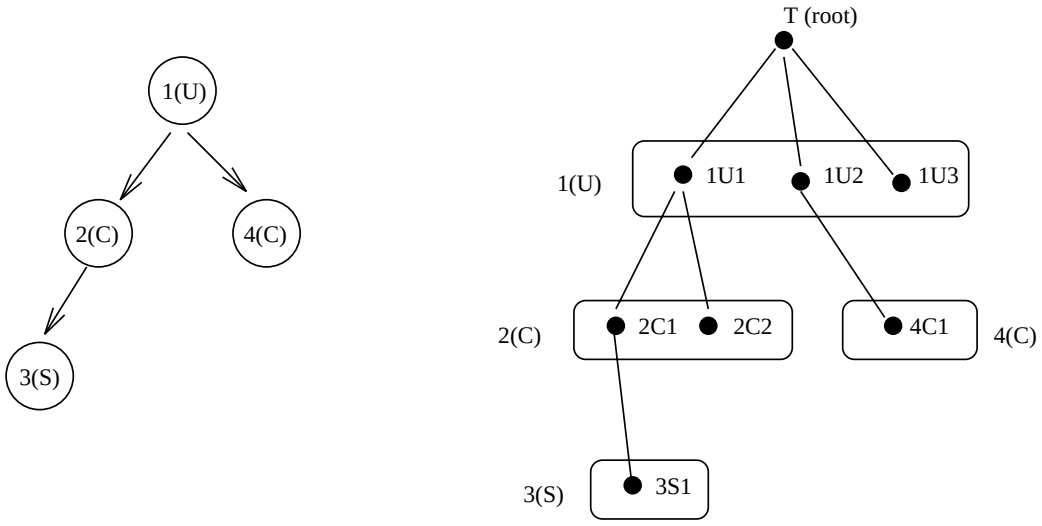

Figure 20: A computation tree and its hierarchical transaction mapping

the correctness of concurrent transactions is governed by serializability. However, if we examine the applications that are impelling the development of object-oriented database technology, we find that they are characterized by requirements that differ from those utilizing conventional databases. These applications are generally found in environments that call for cooperative work (such as computer-aided design). In such environments, serializability as a correctness criterion needs to be relaxed, and interactions between concurrent transactions have to be promoted rather than curtailed. In light of this, in our further discussions we do not assume that serializability is enforced.

We now discuss how a checkin/checkout scheme can be coupled with our hierarchical transaction model so as to facilitate object sharing across concurrent user sessions. Our choice of a checkin/checkout scheme (as opposed to other conventional schemes) directly follows from the above assumption that transactions are cooperative in nature. We provide the following commands to implement a checkin/checkout scheme:

1. **Public-checkout(R/W):** Checks out an object from the public database.
2. **Public-checkin:** Checks in an object into the public database.
3. **Local-checkout(R/W):** Checks out an object from the local workspace of a user session.
4. **Local-checkin:** Checks in an object into a session's local workspace.

The local commands differ from the public ones as their effects are internal to a session, and thus do not affect the visibility or availability of objects to other concurrent user sessions. A checkout operation can be requested in read (R) or write (W) mode. A checkout in W mode is permitted only if the computation generating the requesting subtransaction and the

object to be checked out are at the same level. On the other hand, whenever a computation (or more precisely a subtransaction) requests a checkout of a lower level object, the request is granted in read (R) mode only. Multiple subtransactions may checkout the same object (or version of an object) in R mode. However, if a subtransaction checks out a version in W mode, then no subtransaction may check out the version in either R or W mode (as checkouts in R mode conflict with those in W mode). The W mode checkout operation is thus exclusive with respect to an object. While the checkin operation is necessary for any object checked out in W mode, it is redundant (and can be ignored) for any object checked out in R mode.

If a requested object has not been checked out by a user session so far, a public checkout request is issued. If however the object had been previously checked out from the public database by the session, it is simply checked out from the session's local workspace. In either case, when the subtransaction terminates, the object is checked back into the local workspace of the session. A final version of every object that has been updated will eventually be checked back into the public database (as explained in the remainder of this section).

When a subtransaction succeeds in checking out a version of a lower level object, it is guaranteed that the state of the object so read will never be invalidated in the future. This is because once a version becomes available for checkout in R mode to higher level subtransactions, we are guaranteed that such a version will never be updated again. To put it another way in transaction processing terminology, a checkout in R mode will always read *committed* values of objects. The implication of this is basically that high level subtransactions cannot develop *abort dependencies* on lower level ones. If such dependencies were possible, then a high level subtransaction would have to abort if a low level subtransaction from which it read, aborts.

We now give two variations of a checkin/checkout scheme. They differ basically in how and when objects checked out from the public database are checked back into the public database, for access by other user sessions. They thus offer different granularities of interactions across user sessions. These variations can be applied to both conservative and aggressive intra-session scheduling strategies. In a *level-by-level checkin/checkout* variation, an object that is updated at a level l by a session, is made visible to another session only when all updates to all objects at level l, by the session, have been completed. In the second *computation-by-computation checkin/checkout* variation, an object is made visible (checked in) as soon as all the subtransactions associated with a computation have terminated.

As mentioned before, serializability is *not* enforced across user sessions. However, a subtransaction in a session will see only committed states of objects that are updated by other sessions. This is ensured by requiring all public checkin operations from a session to be deferred until the root computation in the session terminates. We consider a session to be logically and semantically committed at the point the root computation terminates normally (i.e., not due to an error or exception). This guarantees that no *abort dependencies* will develop across user sessions. The absence of such dependencies ensures that a session A would not have to abort because another session B from which it read, aborts.

6.2.1 Level-by-level checkin/checkout schemes

The basic idea is to checkin (commit) objects to the public database, one level at a time. Thus conceptually, we can implement this with processing and propagation of a *level-has-committed* message upwards in the security lattice. With a conservative scheduling scheme, the *level-has-committed* message can be piggybacked onto the *wake-up* message. On the other hand, with aggressive scheduling, the *level-has-committed* message has to be explicitly propagated. We describe both variations below.

The level-by-level checkin/checkout scheme can be combined with the conservative scheduling strategy as follows:

1. A subtransaction checks out the required objects from either its session's local workspace, or from the public database (the latter if any required object has not been previously checked out by the session).

2. When a subtransaction terminates all checked out objects are checked back in to the session's local workspace.

3. If a *wake-up/level-has-committed* message has been received from all immediate lower levels, and all computations and associated subtransactions at a level say l, have terminated (i.e., when the level manager at l finds its local queue to be empty), then the level manager at l checks in the latest versions of all updated objects into the public database. This is followed by step 4.

4. After all updated objects at level l have been checked into the public database, a *wake-up/level-has-committed* message is sent to all immediate higher levels by the local level manager at level l.

In the conservative scheme above, the receipt of a *wake-up/level-has-committed* message from a lower level is a guarantee that no fork requests will be forthcoming from the lower level. However, in an aggressive level-by-level scheme, this is no longer true. In fact, a level may receive many wake-up messages from a lower level. A *level-has-committed* message can thus no longer be piggybacked onto a *wake-up* message, but rather has to be explicitly propagated, starting with the termination of the root computation. In addition to steps (1) and (2) given above for the conservative scheme, we require the following additional steps to achieve this:

3'. When the root computation terminates, we check in all updated objects to the public database and send a *level-has-committed* message to all immediate higher levels.

4'. When a level-has-committed message has been received from all immediate lower levels, and the local level manager finds its queue to be empty, it checks in all updated objects to the public database. The level manager then propagates the *level-has-committed* message to its immediate higher levels.

6.2.2 Computation-by-computation checkin/checkout schemes

A computation-by-computation checkin/checkout scheme releases (checks in) objects to the public database much earlier in comparison to the level-by-level scheme. Thus on the average, the availability of objects for checkout, across user sessions, is increased (as waiting times are reduced). The scheme can be combined again with both conservative and aggressive scheduling. In either case the basic idea is the same. All objects checked out by a computation (the set of subtransactions generated by the computation) are checked back into the *public* database as soon as the computation terminates. Contrast this with the level-by-level checkin scheme where we have to wait for all computations at the associated level to terminate. In other words, when the last subtransaction associated with a computation terminates, all checked out objects are checked back into the public database. However for objects checked out in W mode, only the latest version of every object is checked back in. It is obvious that this variation can result in objects being shuffled back and forth from the public database with much greater frequency than the level-by-level scheme.

7 Summary and Conclusions

In this paper we have discussed an approach to securely and correctly support write-up in terms of abstract operations, within a kernelized architectural framework. Our solution is novel in that it meets the conflicting goals secrecy, integrity, and efficiency. The major complication arises due to the non-primitive nature of such operations. We have discussed an asynchronous computational model that calls for concurrent computations to be generated to service RPC-based write-up requests, and a multiversioning approach to synchronizing such concurrent computations. Although our solution is fitting for object-oriented databases and cast in that context, it is important to emphasize that it has wider applicability in any environment that needs to support write-up operations.

The confidentiality of the scheduling schemes that we have presented is not a concern in a kernelized architecture. However, these scheduling schemes are all inherently secure in that they cannot introduce signaling channels. Hence they can be easily implemented under architectures that are not truly kernelized (i.e., there exists some degree of trust).

In contrasting our work with other proposals for enforcing mandatory security in multilevel object-oriented systems, we note that while they all address confidentiality, the dimension of integrity is largely ignored. Many of the other solutions assume that the TCB provides protection against signaling channels. But how will the TCB do this? Solutions which are implementation dependent are highly vulnerable to the changes and evolution of computer hardware and performance characteristics. Even if timing channels were closed, without synchronization the integrity problem remains unsolved. We believe it is not possible to coin a complete implementation independent solution without the rigor and detail that we have discussed in this paper.

There are several avenues that require further investigation. The multiversioning scheme holds promise for optimizations to reduce the number of versions that need to be concurrently maintained within a user session. Some of these optimizations can be trivially implemented. For example, there is no need to create a new version of an object with the issuing

of a fork, if the object state has not been updated since the last fork request. Our focus in this paper has been to demonstrate the feasibility of a solution. Any implementation must consider the many optimizations possible.

Our approach to session management does not address the issue of atomicity of user sessions in the presence of failed computations. If we take the view that the actions of individual (single-level) computations belong to individual subtransactions, then a user session is analogous to what has been referred to in the literature as a *multilevel transaction*. Such a transaction consists of individual single-level subtransactions. The need for multilevel transactions arises when users have to read and write data classified at multiple levels. Perhaps a good example (and one given in [18]) is a **transfer transaction** in a bank that transfers money from a low-level account to a higher one. Such a task cannot be accomplished by a single-level flat transaction as mandatory access control rules will not permit it to read and write both accounts. Further, this transfer task has to be atomic. Initial investigations of multilevel transactions can be found in [4, 5]. Perhaps the most significant result is the observation in [18] that atomicity and security are conflicting properties. This is because ensuring atomicity will always open up covert channels. The authors in [18] argue for a compromise that limits the bandwidth of such channels in the course of ensuring atomicity.

Closely connected with the above atomicity requirement is the issue of recovery itself, within the context of object-oriented systems and applications. In order to incorporate recovery management semantics into our computational model, we would have to consider the distributed nature of computations as well as the implications of multilevel security. Some of the specific issues/questions that need to be addressed include: (a) How can distributed recovery be incorporated and managed? (b) How can we handle recovery at high levels without interference to lower level computations? (c) What are good granularities for recovery units? Is it a transaction? Is it the state of an object? (d) Can we provide variable granularities of recovery units, so that the amount of loss of work that is tolerable, can be tailored to individual application needs?

A performance evaluation of the various scheduling schemes for computations with varying input-output (I/O) and CPU demands would be interesting. Also interesting is the potential to exploit intra-level concurrency. In both the conservative and aggressive scheduling schemes, there exists only one active computation at a security level, at any given time. This restriction may be relaxed at the cost of managing intra-level conflicts and concurrency. In particular, updates from multiple concurrent computations (at the same level) may have to be serialized at object boundaries. Alternatively, object level semantics may be exploited to allow non-serializable behavior. Such enhancements will impact the level manager algorithms.

The hierarchical model of transactions highlighted in this paper is indeed a primitive one. In particular, it is tightly coupled to our asynchronous computational model. A more advanced object sharing and transaction model should reflect at a more semantic level, the various models of user activity and cooperation in multilevel secure systems (such transaction models for non-secure environments are discussed in [1, 10]). Also, such a transaction model should appear from a user's perspective, to be independent of our underlying asynchronous computational model. Implementing this may naturally lead to a layered transaction scheme, with the higher semantic layer mapped to a lower layer that reflects

the underlying computational model. Another direction worth investigating is a hierarchical model of transactions defined across all user sessions, and to extend the aggressive scheduling strategy across the computations in the various sessions. However, the algorithms in the current form would allow one active session per security level. Hence these algorithms would have to be extended to distinguish computations originating from various sessions as well as to exploit more intra-level concurrency.

References

[1] F. Bancilhon, W. Kim, and H. F. Korth. A model of CAD transactions. *Proc. of the VLDB conference*, 1985, Stockholm, pp. 25-33.

[2] D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, The MITRE Corporation, 1976.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publ. Co., Inc., Reading, MA, 1987.

[4] O. Costich and J.McDermott. A multilevel transaction problem for multilevel secure database systems and its solution for the replicated architecture. *Proc. of the 1992 IEEE Symposium on Security and Privacy*, pp. 192–203, May 1992.

[5] O. Costich and S. Jajodia. Maintaining multilevel transaction atomicity in MLS database systems with kernelized architecture. *Proc. of the 6th IFIP Working Conference on Database Security*, Vancouver, Canada.

[6] D. Fisherman. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):pp. 48–69, January 1987.

[7] S. Jajodia and B. Kogan. Integrating an object-oriented data model with multi-level security. *Proc. of the 1990 IEEE Symposium on Security and Privacy*, May 1990, pp. 76–85.

[8] T.F. Keefe and W.T. Tsai. Prototyping the SODA security model. *Proc. 3rd IFIP WG 11.3 Workshop on Database Security*, September 1989.

[9] T.F. Keefe, W.T. Tsai, and M.B. Thuraisingham. A multilevel security model for object-oriented systems. *Proc. 11th National Computer Security Conference*, October 1988, pp. 1–9.

[10] W. Kim, R. Lorie, D. McNabb, and W. Plouffe. A transaction mechanism for engineering design databases. *Proc. of the VLDB conference*, 1984, Singapore, pp. 355-362.

[11] W. Kim et al. Features of the ORION object-oriented database system. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley Publ. Co., Inc., Reading, MA, 1989.

[12] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proc. 1st Intl. Conf. on Object-Oriented Programming Systems, Languages and Applications*, ACM, New York, 1986, pp. 214-223.

[13] R. Lorie and W. Plouffe. Complex objects and their use in design transactions. In *Proc. Databases for Engineering Applications*, Database Week, 1983 (ACM), May 1983, pp.115-121.

[14] T.F. Lunt. Multilevel security for object-oriented database system. *Proc. 3rd IFIP WG 11.3 Workshop on Database Security*, September 1989.

[15] D. Maier. Development of an object-oriented DBMS. In *Proc. 1st Intl. Conf. on Object-Oriented Programming Systems, Languages and Applications*, ACM, New York, 1986, pp. 472-482.

[16] D. Maier. Why isn't there an object-oriented data model? In *Proc. of the 11th IFIP World computer conference*, San Francisco, CA, August-September, 1989, pp. 793-798.

[17] B. Maimone and R. Allen. Methods for resolving the security vs. integrity conflict. In *Proc. of the fourth RADC Database Security Workshop*, Little Compton, Rhode Island, April 1991.

[18] A. G. Mathur and T.F. Keefe. The concurrency control and recovery problem for multilevel update transactions in MLS systems. *Proc. of the The Computer Security Foundations Workshop VI*, Franconia, New Hampshire, June, 1993, pp. 10–23.

[19] J.K. Millen and T.F. Lunt. Security for object-oriented database systems. In *Proc. of the 1992 IEEE Symposium on Security and Privacy*, May 1992, pp. 260–272.

[20] M. Morgenstern A security model for multilevel objects with bidirectional relationships. *Database Security IV, Status and Prospects*, S. Jajodia and C.E Landwehr (Editors), Elsevier Science Publishers B.V. (North-Holland)

[21] R.S. Sandhu, R. Thomas, and S. Jajodia. A Secure Kernelized Architecture for Multilevel Object-Oriented Databases. *Proc. of the IEEE Computer Security Foundations Workshop IV*, June 1991, pp. 139-152.

[22] R.S. Sandhu, R. Thomas, and S. Jajodia. Supporting timing-channel free computations in multilevel secure object-oriented databases. *Proc. of the IFIP 11.3 Workshop on Database Security,* Sheperdstown, West Virginia, November 1991.

[23] E. Sciore. Object Specialization. *ACM Trans. on Information Systems*, Vol.7, No. 2, April 1989, pp. 103-121.

[24] R.K. Thomas and R.S. Sandhu. Implementing the message filter object-oriented security model without trusted subjects. *Proc. of the IFIP 11.3 Workshop on Database Security*, Vancouver, Canada, August 1992.

[25] M.B. Thuraisingham. A multilevel secure object-oriented data model. *Proc. 12th National Computer Security Conference*, October 1989, pp. 579–590.

[26] D. Ungar, and R. Smith. Self: The power of simplicity In *Proc. 1st Intl. Conf. on Object-Oriented Programming Systems, Languages and Applications*, ACM, New York, 1986, pp. 214-223.