Proceedings of the OOPSLA-93 Conference Workshop on Security for Object-Oriented Systems, Washington DC, September 26, 1993

Concurrency, Synchronization, and Scheduling to Support High-assurance Write-up in Multilevel Object-based Computing

Roshan K. Thomas and Ravi S. Sandhu[1]
Center for Secure Information Systems
&
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030-4444

Abstract

We discuss concurrency, synchronization, and scheduling issues that arise with the support of high-assurance RPC-based (synchronous) write-up actions in multilevel object-based environments. Such environments are characterized by objects classified at varying security levels (called classifications) and accessed by subjects with varying security clearances. A write-up action occurs when a low level object sends a message to a higher one, triggering an update in the latter. While such actions do not directly violate the security policy, their abstract nature in object-based systems poses confidentiality leaks by opening up signaling channels. We present an approach to closing such channels by executing the methods in the sender and receiver objects concurrently, whenever a write-up action is issued. However, these concurrent computations have to be synchronized and scheduled so that they preserve the semantics of the original and synchronous (sequential) execution. We utilize a multi-version synchronization scheme and various scheduling strategies to achieve this.

1 Introduction

We are currently investigating support for secure and efficient RPC-based write-up actions in multilevel object-based computing environments. Such environments are characterized by objects classified at varying security levels (also called classifications or access classes) and accessed by subjects with varying security clearances. These security levels form a lattice structure and mandatory access control is governed by a security policy. The notion of multilevel security originated in the 1960's when the U.S. Department of Defense wanted to protect classified information processed by computers. The Bell-LaPadula (BLP) [1] security model was the first one formally used to implement the military security policy, and even today remains the de facto standard. BLP characterizes (and governs) access control and information flow with the following two rules (l denotes the label of the corresponding subject (s) or object (o)).

- **Simple Security Property.** Subject s can read object o only if $l(s) \geq l(o)$.
- **$\star$-Property.** Subject s can write object o only if $l(s) \leq l(o)$.

[1]The work of both authors is partially supported by a grant from the National Security Agency, contract No: MDA904-92-C-5140. We are grateful to Pete Sell, Howard Stainer, and Mike Ware for their support and encouragement.

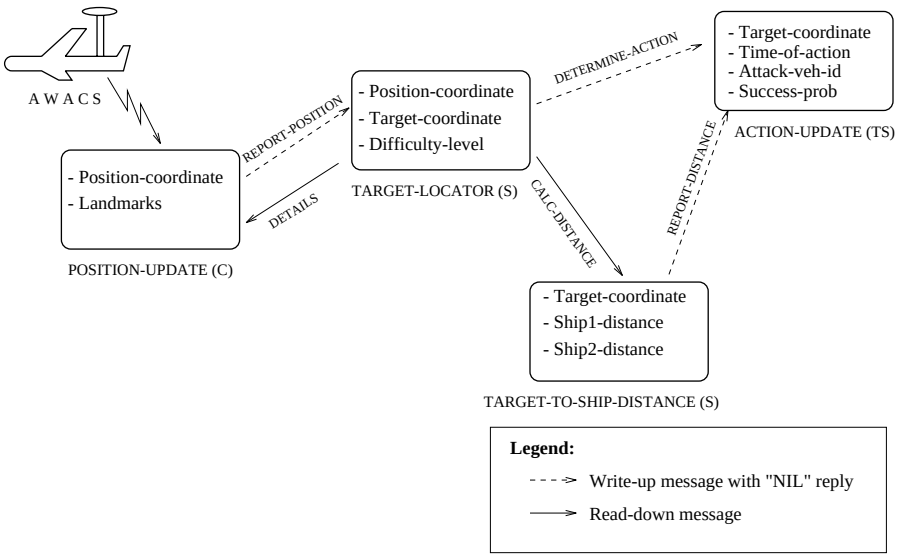

Figure 1: Write-up in situation assesement

In a nutshell, the BLP rules boil down to the fact that a low subject cannot read a high object (called a read-up) and a high subject cannot write a low object (called a write-down). But the BLP mandatory rules do allow write-up actions whereby a low level subject can initiate an update in a high object. However, it is interesting to note that most multilevel (ml) systems such as relational ml DBMS's typically do not allow write-up, due to integrity problems arising from the blind nature of write-up operations in these systems.[2] In object-based computing environments, 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. Further, write-up operations are very useful in many applications.

Figure 1 illustrates a simple situation assessement application in the military setting. The messages REPORT-POSITION, REPORT-DISTANCE, and DETERMINE-ACTION initiate write-up actions. For example, the sending of the REPORT-POSITION message from the lower object POSITION-UPDATE (confidential) to the higher object TARGET-LOCATOR (Secret) results in the invocation of a method that locates a target and updates the attribute 'Target-coordinate' in the latter. It is important to understand why the above cannot be implemented neatly with read-down operations. In applications such as battle management and process control, processing is often initiated by triggered events in the environment. In such scenarios it is difficult to determine the correct polling window for read-down operations. In our example, the higher object TARGET-LOCATOR would have to periodically poll the lower object POSITION-UPDATE for updates in the aircraft's position. If it polls too slowly, the object may miss many position updates. If it polls too frequently the lower object may be inundated with read-down requests, causing considerable processing overhead, and in extreme cases may not be able to keep up with the vital updates from the aircraft. In either case, position updates from the aircraft will be missed with the disastrous consequence that many targets may go unidentified.

However, supporting write-up operations in object-based systems is complicated by the fact that such operations are no longer primitive read's and write's; but can be arbitrarily complex

[2] A good discussion of the conflict between integrity and confidentiality can be found in [4].

and therefore can take arbitrary amounts of processing time. Dealing with the timing of write-up operations consequently has broad implications on confidentiality (due to the possibility of signaling channels), integrity, and performance. Consider what happens when a write-up is initiated and methods (computations) are executed serially. Thus the method in the sender object is suspended until the method in the receiver object has finished executing and returns a reply. Now in the security context, the actual contents of the actual reply from the receiver object cannot be to the lower object (this will be prevented by trusted computing base/security kernel). However we may return an innocuous 'nil' reply. But the very timing of the reply can be observed by the low level suspended method when it is resumed, and this information can be exploited for signaling channels [5]. These channels form means through which high-level subjects can leak information to cooperating lower level subjects. It is important to address these channels since it is well known that mandatory access controls do not provide any protection against them.

We are currently investigating an asynchronous computation model to handle write-up actions. This requires concurrent computations (methods) to be generated whenever write-up actions are issued, and for them to be scheduled and synchronized so that the net effect is logically that of a sequential computation (mimicing RPC semantics). In other words, after sending a message to a higher object, the method in the lower sender object continues executing. The method in the receiver object is executed by a newly created *message manager* process. In this way, the signaling channel is closed as a lower level object never has to wait for a higher one. Our work utilizes an underlying message filter security model to enforce basic mandatory confidentiality [3, 6].

While concurrent computations can close signaling channels, we now have to pay the price of providing synchronization. Synchronization is required to ensure that the concurrent computations have the same effect as the intended serial execution. When this is guaranteed we say that the concurrent computations preserve *serial correctness*. Further, we must ensure that synchronization itself does not cause confidentiality leaks (for otherwise we would be chasing our own tail). To see how serial correctness could be violated, consider again the REPORT-POSITION message in figure 1. Now suppose on receiving the message, the receiver object TARGET-LOCATOR requests additional information about the position identified (such as 'Landmarks') in a read-down message DETAILS. In a serial execution, the DETAILS message would correctly retrieve the required information from the lower object POSITION-UPDATE. This is because processing in this object is temporarily suspended and thus its state would not have been updated after the sending of the REPORT-POSITION message. Contrast this with a concurrent execution, where before the arrival of the DETAILS message, the object POSITION-UPDATE may have updated its state (as it is no longer suspended). Hence DETAILS if allowed to retrieve the 'Landmarks' attribute, would erroneously obtain the values for some later target. Our solution to this synchronization problem calls for lower level objects to save their states before sending write-up messages. The DETAILS message would now retrieve the version of POSITION-UPDATE object whose state existed before the write-up message REPORT-POSITION was issued.

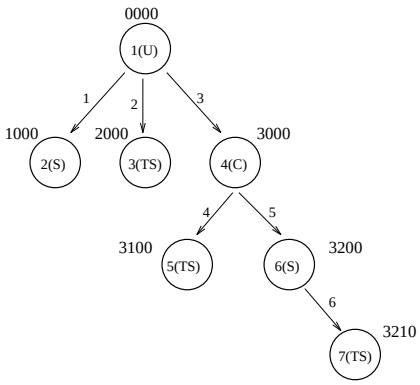

Figure 2: A tree of concurrent computations

2 Concurrency, Synchronization, and Serial Correctness

We now elaborate on concurrency and serial correctness in more general terms. We can visualize a set of concurrent computations as forming a tree such as the one shown in figure 2. The label on the arrows indicate the order in which the messages and the associated computations (methods) would be processed in a serial execution. Note that this order can be derived by a depth-first traversal of the tree. Serial correctness requires that a computation such as 3(TS) in the tree, see all the latest updates of lower level computations to its left, and no updates of lower level computations to its right. Thus 3(TS) should see the latest updates of 2(S) but not of 4(C) and 6(S). This is achieved in our multi-version synchronization scheme by making sure that the versions at levels C (confidential) and S (secret) that are available to 3(TS) are the ones that existed before 4(C) and 6(S) were created (forked). Further, serial correctness also mandates that a computation such as 3(TS) not get ahead of earlier forked ones to its left. Thus 3(TS) should not be started until 2(S) and its children (if any) have terminated.

If no system component has a global snapshot (such as that embedded in a tree) of the entire set of computations, then we need to explicitly capture the global serial order of messages and computations. This can be done by a scheme that assigns a unique forkstamp to each computation, as shown in figure 2. Starting with an initial forkstamp of 0000 for the root, every subsequent child of the root is given a forkstamp by progressively incrementing the most significant digit of this initial stamp by one. To generalize this for the entire tree, we require that with increasing levels, a less significant digit be incremented.

We can now succinctly state the requirements for serial correctness in terms of the following constraints that need to hold 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 fork-

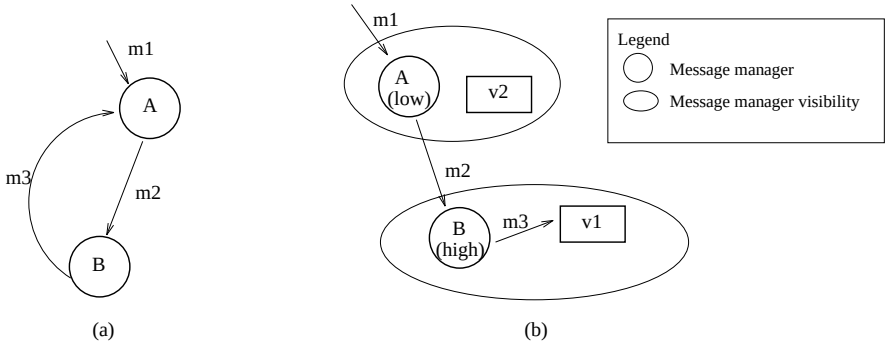

Figure 3: Handling recursion

stamp 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.

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.

Discussion

In concluding this section on concurrency and synchronization, it is important to note that approaches to synchronization across multiple security levels are heavily influenced by mandatory security rules. These approaches thus differ (as they pose a different set of problems) from existing synchronization mechanisms proposed in the literature for general (non multi-level) object-based computing. For example, in asynchronous message passing, the sender and receiver execute concurrently and when the sender needs to access the reply to the message, synchronization is achieved through the notion of futures. When the sender wants the reply from the receiver, it pauses and accesses a *future* data structure to which the receiver would have returned a reply. In the multilevel context, there is no need for futures as a low level sender can never actually access the reply from the higher receiver since this violates mandatory security.

In [2], the authors describe an execution model for distributed object-oriented computation, where methods execute concurrently. To provide synchronization, an approach based on discrete-event simulation is pursued. They describe an interesting scenario with recursion where it may not be possible to finish processing one message before beginning another. As shown in figure 3(a), a message $m1$ is sent to object A, which in turn sends a message $m2$ to object B. A cyclical wait occurs, when B in processing the $m2$ message, sends another message $m3$ back to object A. More precisely, A cannot complete the processing of $m1$ until B completes its own processing of $m2$, and B cannot do this until A completes the processing of message $m3$. Further, if these messages are processed sequentially, serial correctness requires the following: (1) object A should process the $m3$ message with its state not reflecting any of the updates after the message $m2$ was sent; (2) when object A continues processing $m1$ after sending $m2$, its state should reflect all updates intitiated by message $m3$.

How is the above scenario handled in our model when methods execute at different security levels? To start with, we note that a low level object such as A will never have to wait for the

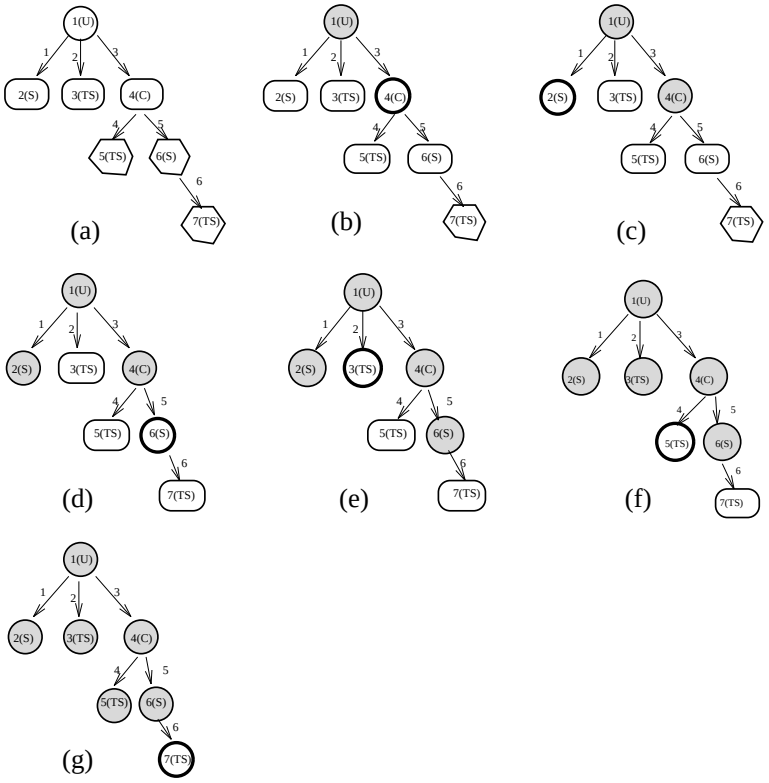

Figure 4: Conservative Scheduling

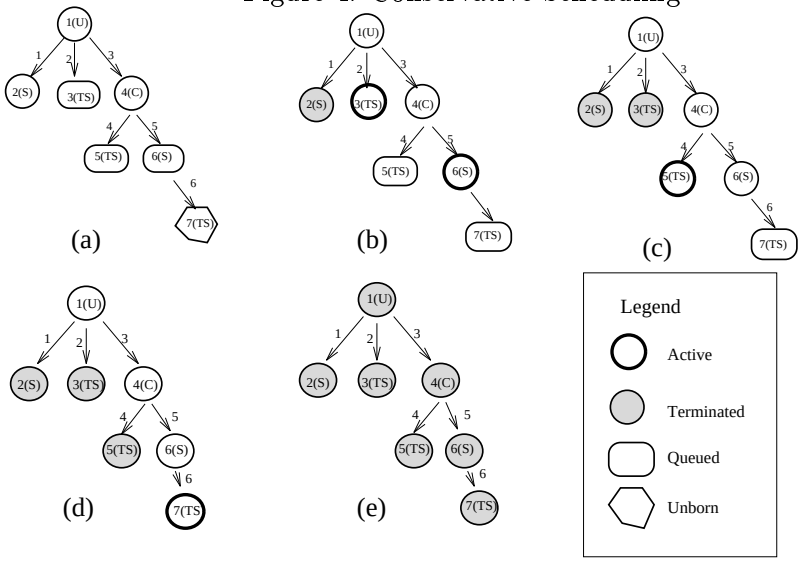

Figure 5: Aggressive Scheduling

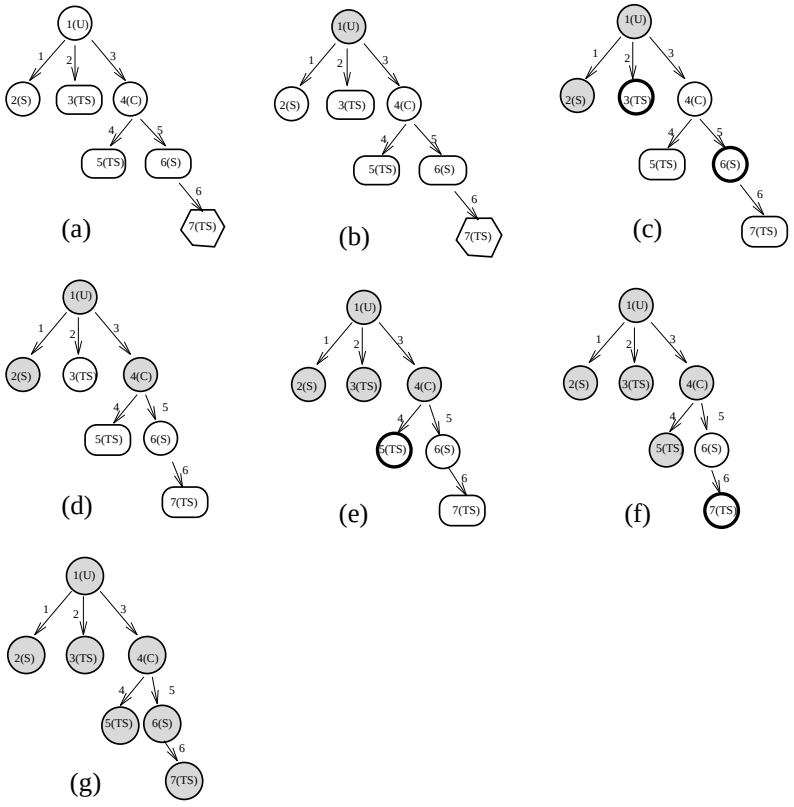

Figure 6: Hybrid Scheduling

processing of message $m2$ sent to the higher level object B. As soon as $m2$ is sent, an immediate reply is returned and A continues processing. The second requirement (listed above) for serial correctness, is trivially satisfied since the message $m3$ can only be a read-down message and can never update the state of object A. The first requirement is satisfied by requiring this read-down message $m3$ to retrieve an older version of object A that existed before $m2$ was sent. This is illustrated in figure 3(b) where object A is required to archive its state as a version ($v1$) just before the write-up message $m2$ is issued. The message manager for the method in object A resumes execution but now accesses and updates a new version ($v2$) of object A. Meanwhile, on receiving the message $m2$, the message manager created to process this message suspends execution of the method in object B, and executes the method in object A to process message $m3$. However it now accesses (reads) the older version of ($v1$) of object A.[3]

3 Scheduling Concurrent Computations

From the above discussion it should be clear that we need to enforce some discipline on concurrent computations as arbitrary concurrency makes synchronization difficult and could lead to the violation serial correctness (thereby affecting the integrity of objects). A scheduling strategy which guarantees serial correctness and at the same time enforces some discipline on concurrency,

[3]Whenever this message manager accesses an object classified lower than the level of object B, it will be prevented from writing to such objects. Otherwise, a write-down violation will occur.

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. The second consideration admits a family of scheduling strategies offering varying degrees of performance. Informally, we say a computation is unnecessarily delayed if it is denied immediate execution on being forked, for reasons other than the violation of serial correctness.

We now consider two scheduling strategies that appear to approach the ends of a spectrum of secure (and correct) scheduling strategies, and a third one that lies somewhere in the middle of such a spectrum. These schemes that lie at the ends of this spectrum are referred to as *conservative* and *aggressive* schemes, and they are governed by the following invariants, respectively.

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.*

Inv-aggressive: *A computation is executing at a level l only if all non-ancestor computations (in the corresponding computation tree) with smaller fork stamps at levels l or lower, have terminated.*

Given a lattice of security levels, the conservative scheme essentially boils down to executing computations on a level-by-level basis in forkstamp order, starting at the lowest level in the lattice. At any point, only computations at incomparable levels can be concurrently executing. However, with the aggressive scheme, we are not following a level-by-level approach. Rather, a forked computation is denied immediate execution only if (at the time of fork) there exists at least one non-ancestral lower level computation with an earlier (smaller) forkstamp, that has not terminated. If denied execution, such a computation is queued and later released for execution when this condition is no longer true (as a result of one or more terminations). A third hybrid strategy can be formulated by executing computations again on a level-by-level basis, but allowing the immediate children of any executing computation to proceed as well. Figures 4, 5, and 6 illustrate the progressive execution of the tree of concurrent computations in figure 2 under the conservative, aggressive, and hybrid strategies, respectively. In each of these figures, the termination of one or more computations (indicated by shaded circles) advances the tree to the next stage. As can be seen in these figures, the tree progresses to termination fastest under the aggressive scheme, since it induces no unnecessary delays. We conjecture that there exists several other variations of the above three scheduling schemes. Finally, it is important to note that the security of these schemes stem from the fact a low level computation is never suspended (delayed) because of a higher one.

4 Summary and Conclusions

In this paper, we have briefly discussed the problem of signaling channels in multilevel object-based computing. The impasse formed by these channels, appears to be fundamental due to the intrinsic abstract nature of operations in object-based computing. We have discussed a solution to close these channels that is based on an asynchronous model of computing and at the price of providing synchronization. Our approach provides a solution that meets the conflicting goals of confidentiality, integrity, and performance. We are currently looking into the scheduling schemes to exploit more intra-level concurrency. In particular, we are investigating how existing general mechanisms can be harmoniously incorporated with the mechanisms to provide synchronization across security levels.

References

[1] D.E. Bell and L.J. LaPadula. Secure computer systems: Unified Exposition and Multics Interpretation. EDS-TR-75-306, The MITRE Corp., Bedford, MA., March 1976.

[2] E.H. Bensley and T.J. Brando and M.J. Prelle. An execution model for distributed object-oriented computation. *Proc. of the ACM OOPSLA conference*, pp. 316–322, September, 1988.

[3] 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*, pp. 76–85, May 1990.

[4] 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.

[5] 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.

[6] 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.