Ensuring Atomicity of Multilevel Transactions*

Paul Ammann Sushil Jajodia
Indrakshi Ray
Center for Secure Information Systems and
Department of Information and Software Systems Engineering
George Mason University
Fairfax, VA 22030-4444
{pammann,jajodia,indrakshi}@isse.gmu.edu

Abstract

Ensuring atomicity is a major outstanding problem with present methods of handling multilevel transactions. The chief difficulty is that a high section of a transaction may be unable to complete due to violations of the integrity constraints, and a rollback of sections can be exploited to implement a covert channel. We define a notion of semantic atomicity which guarantees that either all or none of the sections of a transaction are present in any history. The notion of correct executions in our model is based on semantic correctness – that is, maintenance of integrity constraints – rather than serializability. We give a method whereby the application developer can statically analyze the set of transactions in the application and determine if the set ensures semantic atomicity and other desirable properties.

1. Introduction

Most of the research of transaction processing in multilevel secure databases is limited to single-level transactions. A fixed security level is associated with a single-level transaction, which can read objects that are at its level or below, but can only write those objects that are at its level.† These restrictions correspond to the simple security property and the restricted $\star$-property of the Bell-LaPadula model [3] and ensure that no illegal direct information flows occur across security levels.

By definition, a single-level transaction prevents operations at different levels from being grouped as an atomic transaction. This is a serious limitation in many applications. Consider the example, adapted from [6, 7], of a satellite that gathers and writes various data in a multilevel secure database. The satellite records its *position data* (which is unclassified), gathers and then records its *sensor data* (which is sensitive), performs some analysis on the basis of these data and finally writes a *result* (which is again sensitive). Modeling such an application with a decomposition into a set of independent single-level sections is undesirable for two reasons. First, it forces the user to manage the scheduling of the single-level sections. Even worse, interleavings with other transactions may produce incorrect results. Consequently, multilevel transactions [5, 6, 7, 12] have been proposed to overcome this difficulty. A multilevel transaction permits read and write operations across a range of security levels to be executed as an atomic unit. Atomicity ensures that after the completion of a multilevel transaction either all its changes persist or none do.

To limit the size of the trusted code, a multilevel transaction is usually decomposed into a collection of single-level subtransactions called sections. The interactions between the sections can be analyzed to determine the presence of any illegal information flows across security levels [5]. If there are no illegal information flows, then the sections may be executed as atomic units. Costich and McDermott describe a transaction processing algorithm for the replicated architecture in [7], and Costich and Jajodia give one for the kernelized architecture in [6]. A limitation of both these works, however, is that they are based on the optimistic assumption that if any low section commits, all other sections in the multilevel transaction must also commit (see Section 3 for a complete discussion).

Unfortunately, ensuring atomicity of a multilevel transaction from the atomicity of sections is not an easy task. Consider a multilevel transaction T_i which is decomposed

*This work was partially supported by grants from the National Science Foundation grant numbers CCR-9202270 and IRI-9303416, by a grant from ARPA, administered by the Office of Naval Research under grant number N0014-92-J-4038, and by National Security Agency under contract number MDA904-94-C-6118.

†Keefe and Tsai [10] use a slightly more general model in which the single-level transactions are allowed to write to objects that are at or above its level.

into two sections, S_{i1} which is *low*, and S_{i2} which is *high*. Suppose S_{i1} executes successfully, but S_{i2} fails. Atomicity requirements force us to either (i) abort S_{i1} or (ii) reexecute S_{i2}. Since aborting S_{i1} may establish a covert channel between the high section S_{i2} and the low section S_{i1}, we are left with only one option – reexecute S_{i2}. However, attempts to reexecute S_{i2} may fail again. For example, the preconditions of S_{i2} may not be satisfied because of integrity constraint violations. The point is that the atomicity of the multilevel transaction T_i cannot be ensured.

Blaustein et al. [5, 12] define different degrees of atomicity – ML atomicity, L-atomicity, and complete atomicity. The most stringent, complete atomicity, corresponds to traditional atomicity. [5] also gives two transaction processing algorithms – High-Ready-Wait and Low-First. The High-Ready-Wait algorithm is a two-phased algorithm. In the first phase the sections of a multilevel transaction are executed (but not committed) in a high to low order; in the second phase the sections are committed in the low to high order. The High-Ready-Wait suffers from three problems. First, it cannot handle multilevel transactions having low to high dependencies. Second, it is suitable only for those multilevel transactions in which the security levels of the sections are linearly ordered; that is, it cannot handle multilevel transactions in which two sections have incomparable security levels. Third, it contains a limited bandwidth timing channel; a high transaction can modulate the time it takes to execute and thereby convey information to a low section. Unlike High-Ready-Wait, Low-First executes and commits sections of a multilevel transaction in a low to high order. The limitation of Low-First is that it cannot make any guarantees with respect to the atomicity of the multilevel transaction.

All previous work on multilevel transactions [5, 6, 7, 12] relies on the standard conflict-serializability [4] as the correctness criterion. The problem with these approaches is that complete atomicity cannot be guaranteed. Our contribution is to propose an alternative semantic-based model for multilevel secure transactions. Our model uses the notion of multilevel semantic histories to reason about correct interleavings of sections. In particular, we develop three properties, the consistent execution property, the sensitive transaction isolation property and the semantic atomicity property. The consistent execution property ensures that if a complete semantic history executes from a consistent state, the final state is also consistent. The sensitive transaction isolation property ensures that no inconsistencies are displayed to the user. The semantic atomicity property ensures that all partial semantic histories can eventually complete, or, in other words, either all or none of the sections of a transaction appear in a history. Our work builds on research from outside the security domain where semantic-based concurrency control has been used to improve performance for applications with long-duration transactions [2, 8, 9, 11].

$\underline{room}$	$status$
Room 1	$Available$
Room 2	$Unavailable$
⋮	⋮
Room 128	$Unavailable$

(a) Table $STATUS$

$\underline{room}$	$guests$
Room 2	Lafayette
⋮	⋮
Room 128	Hemingway

(b) Table $GUEST$

Figure 1. Tables for hotel example

The organization of the paper is as follows. The next section outlines an example that is used to illustrate the rest of the paper. Section 3 explains the advantages of a semantic approach to atomicity. Section 4 gives basic definitions for multilevel transactions, and section 5 defines semantic histories and semantic atomicity. Section 6 discusses some implications of our model and concludes the paper.

2. Explanatory Example

Example 1 [Basic Hotel] We use a running example of a hotel database to explain our ideas. We chose the hotel example because its semantics are intuitively clear and it makes a good basis for illustrating our points. We do not sacrifice relevance or applicability in this choice; in section 6 we outline a military-oriented example that has exactly the same formal structure as the hotel example.

The hotel database has a set of rooms. The status of each room is either $Available$ or $Unavailable$. This information is stored in a table $STATUS$ with two attributes - $room$ and $status$. The attribute $room$ is the primary key for the table $STATUS$. The knowledge of whether a room is available or not is known by everyone, and hence table $STATUS$ has security level low. The table $GUEST$ maintains information about which room has been assigned to which guest. $GUEST$ has two attributes - $room$ and $guest$. The attribute $room$ is the primary key for the table $GUEST$. The set of guests in the hotel is sensitive information and so table $GUEST$ has security level high. The structure of the tables are shown in figure 1.

The hotel database has the following integrity constraints:

1. Each room has only one status – $Available$ or $Unavailable$,
2. Each room can be assigned to one guest, and
3. The rooms with status $Unavailable$ have been assigned to guests.

The first two integrity constraints follow from the definition of primary key. The third integrity constraint implies that all rooms with status $Unavailable$ in table $STATUS$ also appear in the $room$ field in table $GUEST$. Note that the third integrity constraint is on objects at different security levels, a situation that can hardly be avoided. Integrity constraints depend on the semantics of the application and are generally unrelated to the assignment of security levels to data; hence the application developer should expect that integrity constraints will span security levels and take appropriate steps to ensure that such constraints are maintained in a secure manner.

There are three types of transactions in the hotel database – $Reserve$, $Cancel$ and $Report$. The $Reserve$ transaction checks to see if a room is available, selects a room for assignment and assigns the room to the guest. The input to the $Reserve$ transaction is g - the guest for whom the room is being reserved. The output of the $Reserve$ transaction is r - the room that is assigned to the guest. The $Cancel$ transaction cancels the reservation for a room r. The $Cancel$ transaction checks to see if the status of r is $Unavailable$, changes the status of r to $Available$, and the entry corresponding to room r is removed from the table $GUEST$. The input to the $Cancel$ transaction is r. The $Report$ transaction outputs the contents of tables $STATUS$ and $GUEST$. The details of the sections into which the multilevel transactions are decomposed are described in subsequent examples. □

3. Motivation For Semantic Atomicity

The main contribution of this paper is the development of a notion of semantic atomicity that is secure for multilevel transactions. Before developing such a notion we explain the advantages of a semantic approach over the traditional approach to correctness via conflict serializability.

In considering reasons why transactions may abort, we regard external agents such as the user and processes that manage recovery from site and network failures as not relevant to the contribution of this paper. There remain two reasons internal to the database why a transaction may abort:

1. The transaction cannot complete without violating some integrity constraint.
2. The concurrency control mechanism cannot schedule some operation of the transaction due to interference by concurrent transactions.

The first of the reasons listed motivates the following property on each transaction considered in isolation:

Definition 1 [Isolation Atomicity] A multilevel transaction enjoys the *isolation atomicity* property if for any given consistent state of the database, satisfaction of the precondition of the least[‡] section implies the satisfaction of the preconditions of all sections. If the transaction has no least section, then the precondition for the transaction is that the database is in a consistent state.

Isolation atomicity is a strong requirement, as the following example shows.

Example 2 [A Transaction Without Isolation Atomicity] We modify the basic hotel example by adding the additional integrity constraint that each guest can have at most one reservation. The sections of the $Reserve$ transaction for this modified example are given below. The first section is of level low and the second section is of level high. The database read and write operations are enclosed in brackets.

1. check $STATUS$ to see if any room r has status $Available$ [read $STATUS$]
   update $STATUS$ by changing the status of r to $Unavailable$ [write $STATUS$]
2. check $GUEST$ to see if the guest g has any reservation [read $GUEST$]
   check $GUEST$ to see that room r is not assigned to any guest [read $GUEST$]
   update $GUEST$ to add the entry corresponding to r and g [write $GUEST$]

Suppose this reserve transaction is executed for a guest g who already has a reservation. Section 1 can be executed successfully. However, the first precondition of section 2 is not satisfied and so it cannot be executed. (The second precondition of section 2 is guaranteed to be satisfied by the third integrity constraint in example 1, and so is not a problem.) Thus the $Reserve$ transaction does not have the isolation atomicity property. □

If serializability is the correctness criteria, then the isolation atomicity property is clearly necessary to preclude a transaction from aborting due to an integrity constraint violation. However, as the following example shows, the isolation atomicity property is not sufficient to ensure atomicity if transactions are executed concurrently by a mechanism that guarantees conflict serializable histories.

[‡] A section in a multilevel transaction is the least section if its class is dominated by the classes of all other sections in the transaction. Formal definitions of the security structure appear below in Section 4.

Example 3 [Isolation Atomicity Is Not Sufficient For Complete Atomicity] Consider the $Reserve$ and $Cancel$ transactions for example 1. The $Reserve$ transaction is composed of two sections - the first section has security level low and the second section has security level high. The sections are as follows.

1. check $STATUS$ to see if any room r has status $Available$ [read $STATUS$]
   update $STATUS$ by changing the status of r to $Unavailable$ [write $STATUS$]

2. check $GUEST$ to see that room r is not assigned to any guest [read $GUEST$]
   update $GUEST$ to add the entry corresponding to r and g [write $GUEST$]

If the $Reserve$ transaction is executed in isolation and in a consistent state, the precondition of the second section will always be satisfied. This is because the satisfaction of the preconditions of the first section and the integrity constraints guarantees the satisfaction of the precondition of the second section. In other words, the $Reserve$ transaction in this example has the isolation atomicity property.

The two sections of $Cancel$ transaction are described informally below. The first section has security level low and the second section has security level high.

1. check $STATUS$ to see if room r has status $Unavailable$ [read $STATUS$]
   update $STATUS$ to change status of r to $Available$ [write $STATUS$]

2. check $GUEST$ to see if r has a guest assigned to it [read $GUEST$]
   update $GUEST$ to remove the entry associated with r [write $GUEST$]

The precondition of the second section of the $Cancel$ transaction prevents us from deleting an entry in table $GUEST$ that does not exist. The precondition of the second section of the $Cancel$ transaction holds if the integrity constraints and the preconditions of the first section are satisfied. Thus the $Cancel$ transaction has the isolation atomicity property.

However isolation atomicity does not guarantee complete atomicity using the Low-First algorithm. Suppose a timestamp ordering (TO) is used to generate serializable schedules using the Low-First Algorithm. When a transaction is submitted to the system it is assigned a timestamp. For the sake of simplicity assume that the timestamp of transaction T_i is i. Sections of a transaction have the same timestamp as the transaction. The sections of a transaction are then distributed to the respective single level schedulers. Each single level scheduler is responsible for scheduling the operations of a particular level using the TO algorithm.

Consider the history: $H = < S_{11}, S_{21}, S_{22} >$. S_{11} represents the first section of a $Reserve$ transaction T_1. S_{21} and S_{22} represent the two sections of a $Cancel$ transaction T_2. The timestamp associated with the sections of $Cancel$ transaction is greater than the timestamp associated with the sections of $Reserve$ transaction. Now if section S_{12} arrives after section S_{22} it will not be possible to execute section S_{12}. This is because the write operation (write $GUEST$) in section S_{12} has arrived after the read operation (read $GUEST$) of section S_{22} has already been processed. Thus the timestamp ordering algorithm will abort section S_{12}, and fail to achieve complete atomicity for transaction T_1. $\Box$

The difficulty in the example above is that a high section of one transaction may access data 'too late' and hence must be aborted. It is possible to design concurrency control mechanisms that avoid this problem and thereby guarantees atomicity if each transaction satisfies the isolation atomicity property. The basic requirement on such algorithms is that sections are not aborted for concurrency control reasons.

Costich and Jajodia describe a multiversion timestamping algorithm that satisfies this requirement [6]. The basic idea is that when a section of a transaction begins at a given security level, the set of data elements that the section reads is compared with the set of data elements to be written by transactions with earlier timestamps. When the intersection of these two sets is empty, the transaction is allowed to proceed. The result is that a section S is never too late to write a data item, since any other section that must read the data item from S to maintain serializability is delayed until S commits.

Given isolation atomicity, the Costich and Jajodia algorithm produces histories that are one-copy serializable. However, there are two costs of insisting on conflict serializability as the correctness criteria:

- Sections at dominating levels must be delayed, thereby reducing the amount of concurrency possible.
- Isolation atomicity is necessary.

For some applications one or both of these costs can prove unacceptable. By instead basing the notion of correctness on the semantics of the application, it is possible to increase the amount of concurrency above that offered by requiring conflict serializability. The resulting mechanism allows more histories, and, as a bonus, does not require a mechanism to delay high sections. Instead, standard concurrency control mechanisms can be used to control interleaving at each security level. In addition, isolation atomicity is no longer a necessary condition, as we demonstrate by example later in the paper (see example 10).

The cost of semantic concurrency control is that the set of transactions in an application must be analyzed off-line to ensure a set of properties we derive and explain in the remainder of the paper. However, this is a one-time cost incurred during the specification stage of an application, and not a performance penalty that must be endured during every execution. If the properties are satisfied, then semantic atomicity is ensured. If the properties are not satisfied and atomicity is required, then the application must be revised. Note that some requirements inherently make atomicity unachievable in a secure manner whether the correctness criteria is serializability or semantic correctness. For example, the requirement that guests cannot register multiple times in example 2 falls into this category.

4. Definitions

We consider a security structure that is a partial order, (**C**, $<$). **C** is a set of security levels or classes, and $<$ is the dominance relation between classes. Two classes C_1 and C_2 are said to be incomparable if neither $C_1 \leq C_2$ nor $C_2 \leq C_1$. Each database object $x \in \mathbf{D}$ is associated with exactly one security level. However, for the reasons outlined in the introduction, we permit a transaction to read and write data at a range of levels.

Definition 2 [Multilevel Transaction] A multilevel transaction T_i is a set of read and write operations in which conflicting operations [4] are ordered by the relation $\ll_i$.

To prevent direct violations of the usual mandatory access control policy [3] by a multilevel transaction, we follow other authors [7, 6, 12] and decompose each multilevel transaction into a set of *sections*, where each section is associated with a single security level. To keep our model simple, we assume that a transaction has at most one section at any security level. Each write operation in a multilevel transaction is associated with exactly one section, but a single read operation in the multilevel transaction may appear in multiple sections. Let the function L map database objects and sections to security levels. We require a section S_{ij} of multilevel transaction T_i to obey the simple security property and the restricted $\star$-property:

1. A section S_{ij} with $L(S_{ij}) = C$ may read a database object x if $L(x) \leq C$.
2. A section S_{ij} with $L(S_{ij}) = C$ may write a database object x if $L(x) = C$.

Based on this we define transaction decomposition.

Definition 3 [Transaction Decomposition] A multilevel transaction T_i is *decomposed* into a set of sections $\{S_{i1}, S_{i2}, \ldots, S_{in}\}$ such that

1. conflicting operations in section S_{ij} are ordered by the relation $\ll_{ij}$,
2. orderings of conflicting operations specified in a section honor the orderings of conflicting operations specified in the transaction, that is, $\ll_{ij} \subseteq \ll_i$, and
3. each write operation in T_i appears in exactly one section and each read operation in T_i appears in at least one section.

To avoid timing channels, it is necessary to order the sections in a way consistent with the security lattice structure (low to high order). However given a transaction decomposed according to Definition 3 it may not always be possible to execute the sections in a low to high order, and at the same time respect the ordering relation $\ll_i$. In the following paragraphs we explain how to solve this problem.

First, we need the notion of dependencies from [5]. A *dependency* exists between two sections if they contain conflicting operations. There can be two types of dependencies between sections of a transaction, depending on the ordering of operations in the underlying multilevel transaction.

1. L-H Dependency (Low to High Dependency) – A high section of T_i reads x, a low section of T_i writes x, and in the multilevel transaction T_i, $w_i[x] \ll_i r_i[x]$
2. H-L Dependency (High to Low Dependency) – A high section of T_i reads x, a low section of T_i writes x, and in the multilevel transaction T_i, $r_i[x] \ll_i w_i[x]$

We require any implementation supporting multilevel transactions to provide the following two properties to correctly treat L-H and H-L Dependencies:

Definition 4 [Internal Reads Property] A multilevel transaction enjoys the *internal reads* property if for each pair of sections with a L-H Dependency, the high section reads the value x written by the low section.

Definition 5 [Read First Property] Sections in a multilevel transaction T_i enjoy the *read first* property if for each pair of sections with a H-L Dependency, the high section reads the value x has immediately before the low section overwrites x[§].

We adopt a caching technique[12] as the mechanism to satisfy the read first and the internal reads property. The cache is local to the transaction; it is visible only to sections of a single transaction. In accordance with security requirements, a high section can read the cached values written by the low section but not vice-versa. A section which updates any database object writes the old value and the modified value in the cache. The internal reads property can be satisfied using the cache. Consider a L-H dependency in a

[§] In the case where the multilevel transaction does not write x, we require that all reads of x return the same value.

transaction: $w_i[x] \ll_i r_i[x]$ where $w_i[x]$ is an operation in the low section and $r_i[x]$ is an operation in the high section. The low section writes the new value of x in the cache. Subsequently, the high section reads this new value of x from the cache.

The read first property can also be satisfied with the cache. Consider an H-L dependency in a transaction: $r_i[x] \ll_i w_i[x]$ where $r_i[x]$ is an operation in the high section and $w_i[x]$ is an operation in the low section. When the low section executes, it writes the old value and the new value of x into the cache. The database read operation $r_i[x]$ in the high section is converted to a cache read; the high section reads the old value of x from the cache. This technique converts the database read operations present in the high section to cache reads. The caching technique can eliminate all H-L dependencies. Once all the H-L dependencies are eliminated, the only dependencies that exist between sections of a transaction are the L-H dependencies. With only L-H dependencies remaining, it is possible to order the sections of a transaction low first.

Definition 6 [Partial Order $\prec_i$ On Sections In Transaction T_i] Let $\prec_i$ be a partial order defined on the sections of transaction T_i. For any two sections S_{ij},S_{ik} of transaction T_i which are related by the partial order $\prec_i$, $S_{ij} \prec_i S_{ik}$, if all operations of section S_{ij} precede any operation of S_{ik}.

Next we define what it means for a transaction to be in canonical form. Informally a transaction is in canonical form if its sections are ordered according to the security lattice structure.

Definition 7 [Canonical Form] A transaction T_i is in *canonical form* if for all j, k, $S_{ij} \prec_i S_{ik}$ iff $L(S_{ij}) < L(S_{ik})$.

Our canonical form is a partial order $\prec_i$ of sections in contrast to [7] where canonical form is defined as a total order of sections.

The importance of the methods for treating H-L and L-H dependencies is that *any* multilevel transaction can be converted to canonical form and securely executed with the Low-First algorithm. In prior work, this transformation yielded only ML-atomicity [5]; our result is to achieve semantic atomicity, where either all or none of the sections are executed.

5. Semantic Correctness

We define a database from a semantic perspective. A *database* is specified as a collection of objects, along with some *invariants* or *integrity constraints* on these objects. A database is said to be in a consistent state if all the integrity constraints hold; otherwise the database is in an inconsistent state. We do not impose any constraint on the nature of integrity constraints. That is, integrity constraints may be defined over objects belonging to the same level or they may be defined over objects belonging to different security levels. A section of level C can only view those integrity constraints which involve objects at level C or below. Consequently, only the highest section, assuming there is one, can view all the integrity constraints of the database.

Each multilevel transaction is specified with a set of *preconditions* and a set of *postconditions* on the database objects. A precondition limits the database states to which a multilevel transaction can be applied. A postcondition constrains the possible database states after a multilevel transaction completes. Together, preconditions and postconditions must ensure that if a multilevel transaction executes on a consistent state, the result is again a consistent state. The preconditions and postconditions of each section arise from the preconditions and postconditions of the multilevel transaction from which the section is derived.

When a transaction has been partially executed (that is, some but not all of its sections have been committed) the database may be in an inconsistent state.

Example 4 [Inconsistent Intermediate States] Consider the $Reserve$ transaction from example 3. After the execution of the low section of the $Reserve$ transaction the database is in an inconsistent state; the result of the low section is that the third integrity constraint of example 1 no longer holds, namely, there is now an unavailable room that is not assigned to a guest. □

Partially executing a transaction not only violates database consistency but also impacts the isolation property. It is necessary to ensure that each section of a multilevel transaction has preconditions that are strong enough to ensure that the section is correct when confronted with an intermediate state left by the partial execution of some other multilevel transaction. If the section generates output in addition to accessing the database, it is necessary to ensure that the output appears to be generated from a consistent state even if the section executes on an intermediate state. Finally, the integrity constraints must be restored upon the completion of all transactions. To ensure these properties, we develop the notion of a multilevel semantic history.

Definition 8 [Multilevel Sectionwise Serial History] A multilevel sectionwise serial history H defined over a set of transactions $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ is a sequence of sections with the ordering relation $\prec_H$ such that,

1. for each $T_i \in \mathbf{T}$, a section of T_i either appears exactly once in *H* or does not appear at all,

2. for any two sections S_{ij}, S_{ik} of some $T_i \in \mathbf{T}$, $S_{ij} \prec_H S_{ik}$ in *H* if $S_{ik} \not\prec_i S_{ij}$ in canonical form of T_i,

3. if $S_{ik} \in H$, then $S_{ij} \in H$, for all sections S_{ij} such that $S_{ij} \prec_i S_{ik}$ in canonical form of T_i.

By Condition (1), we ensure that every section of a transaction should occur at most once in a multilevel sectionwise serial history. Condition (2) ensures that the order of the sections in canonical form of a transaction should be preserved in the multilevel sectionwise serial history. Condition (3) ensures that for every section S_{ik} in a multilevel sectionwise serial history, all the sections that precede S_{ik} in canonical form of T_i are present in the multilevel sectionwise serial history.

Not all multilevel sectionwise serial histories are correct. To reason about interleavings, we introduce the notion of multilevel *semantic* histories which not only list the sequence of sections forming the history, but also convey information regarding the state of the database before and after execution of each section in the history.

Definition 9 [Partial Multilevel Semantic History] A *partial multilevel semantic* history H is a multilevel sectionwise serial history that is bound to

1. an initial state, and
2. the states resulting from the execution of each section in H.

Definition 10 [Complete Execution] An execution of a transaction $T_i = < S_{i1}, S_{i2}, \ldots, S_{in} >$ in a multilevel sectionwise serial history ***H*** is a *complete* execution if all n sections of T_i appear in H.

Definition 11 [Complete Multilevel Semantic History] A partial multilevel semantic history H over **T** is a *complete* semantic history if the execution of each T_i in **T** is *complete*.

Not all multilevel semantic histories are acceptable. In the next section we identify three necessary properties which allowable multilevel semantic histories should possess.

5.1. Properties of Histories

In a secure environment, desirable semantic histories possess the following three properties.

5.1.1. Consistent Execution Property

An important property of databases is consistency. Each transaction when executed in isolation maintains database consistency. Therefore in transaction processing models using serializability as the correctness criteria, database consistency is always maintained. However, in our model, which uses semantic correctness rather than serializability, database consistency is not automatically ensured. The following example helps to illustrate this point.

Example 5 [Complete Multilevel Semantic History Leaves Database In An Inconsistent State] Consider the satellite example cited in [6, 7]. The satellite report transaction takes as input $position\_data$, and $sensor\_data$. The $position\_data$ indicates the position of the satellite and is of level unclassified **U**. The $sensor\_data$ is the information received from its sensor. The level of $sensor\_data$ is secret **S**. The satellite report transaction records the $position\_data$ and $sensor\_data$ in the database and also performs some analysis on these data. The results of the analysis are stored in the database object $result$. The level of $result$ is secret **S**. For reasons of security, the satellite report transaction is decomposed into two sections; the first one has security level unclassified and the second one has security level secret.

1. input $position\_data$
   record $position\_data$ [write $position\_data$]
2. input $sensor\_data$
   read $position\_data$ [read $position\_data$]
   compute $result$
   record $sensor\_data$ [write $sensor\_data$]
   record $result$ [write $result$]

Suppose there is an integrity constraint involving the database objects $position\_data$, $sensor\_data$ and $result$: $result = fn(position\_data, sensor\_data)$. Consider the following history H composed of two satellite report transactions T_1 and T_2: $H =< S_{11}, S_{21}, S_{22}, S_{12} >$. The application demands that the satellite report transaction have the internal reads property. In other words, the value of $position\_data$ read by the section S_{12} should be the one written by section S_{11} and not the one by section S_{21}. After the history H completes, the database has the following values: $position\_data$ - written by section S_{21} of transaction T_2, $sensor\_data$ - written by section S_{12} of transaction T_1, and $result$ - written by section S_{12} of transaction T_1. Clearly the integrity constraint, defined over the database objects $position\_data$, $sensor\_data$ and $result$, does not hold after the execution of the history H. $\Box$

The above example motivates us to define the consistent execution property on semantic histories:

Definition 12 [Consistent Execution Property] If a complete semantic history is executed in a consistent state, then the final state resulting from the execution of the history is also a consistent state.

Note that any complete multilevel semantic history generated from example 3 has the consistent execution property.

5.1.2. Sensitive Transaction Isolation Property

The partial execution of a multilevel transaction may leave the database in an inconsistent state, and other sections may be exposed to this inconsistency. In some cases this is problematic. For example, a multilevel transaction may output data to users; these transactions are referred to as *sensitive* transactions in [9]. Consider the following example.

Example 6 [Sensitive Transaction Views Inconsistent State] Let us once again consider the basic hotel example. $Reserve$ is as in example 3. The $Report$ transaction is composed of a single high section.

1. output $STATUS$ [read $STATUS$]
   output $GUEST$ [read $GUEST$]

If $Report$ transaction is executed between the two sections of the $Reserve$ transaction, inconsistencies will be displayed to the user. The users will see the violation of the third integrity constraint specified in example 1. (When only the first section of the $Reserve$ transaction is executed, there is a room with status $Unavailable$ that has not been assigned to any guest). □

Definition 13 [Sensitive Transaction Isolation Property] All output data produced by a sensitive multilevel transaction T_i appears to be generated from a consistent state, even though T_i may be executing in an inconsistent state.

To ensure the sensitive transaction isolation property, a section in a sensitive multilevel transaction may require an additional precondition equivalent to some or all of the integrity constraints on objects at dominated levels. In example 6, the output of $Report$ can be ensured to be consistent by strengthening the precondition to explicitly check for the third integrity constraint in example 1.

Example 7 [Satisfaction Of The Sensitive Transaction Property] The $Reserve$ and $Cancel$ transactions are as in example 3. The $Report$ transaction reads the tables $STATUS$ and $GUEST$ and outputs their contents. $Report$ can be written as a single section with security level high, but we choose to decompose the $Report$ transaction into a low and a high section.

1. output $STATUS$ [read $STATUS$]

2. check $STATUS$,$GUEST$ if $Unavailable$ rooms are assigned to guests [read $STATUS$, $GUEST$]
   output $GUEST$ [read $GUEST$]

Note that an extra precondition has been added to the second section of the $Report$ transaction to ensure the sensitive transaction isolation property. □

5.1.3. Semantic Atomicity

When transactions have been broken up into sections, the interleaving of sections may lead to deadlock (that is, a state from which we cannot complete some partially executed transaction). Semantic atomicity ensures that deadlock is avoided; if a transaction has been partially executed, then it can eventually complete.

Example 8 [An Application That Lacks Semantic Atomicity] We modify the example 1 to accommodate a procedure whereby rooms are checked between guests. After a guest cancels a room, the room is inserted in the list of rooms that have to be checked. The list of rooms that have to be checked is kept in a table $CHECK$. The security level associated with this table is high.

We leave the $Reserve$ transaction unchanged from example 3, which, as we will discover later, is a mistake.

The $Cancel$ transaction is modified as follows. An extra operation has been added to the high section of the $Cancel$ transaction. This operation updates table $CHECK$ to insert an entry corresponding to room r. An extra precondition has been added to the high section of the $Cancel$ operation that checks whether r is not in the list of rooms to be checked.

1. check $STATUS$ to see if room r has status $Unavailable$ [read $STATUS$]
   update $STATUS$ to change status of r to $Available$ [write $STATUS$]

2. check $CHECK$ to see if r is not in the list of rooms to be checked [read $CHECK$]
   check $GUEST$ to see if r has a guest assigned to it [read $GUEST$]
   update $GUEST$ to remove the entry associated with r [write $GUEST$]
   update $CHECK$ to insert the entry corresponding to r [write $CHECK$]

There is another transaction type $Check$ which checks a room. The transaction type $Check$ consists of a single high level section. The first precondition ensures that the room to be checked must not be assigned to any guests. The second precondition verifies whether the room is in the list of rooms to be checked. The details are as follows.

1. check $GUEST$ to see if r is not assigned to any guest [read $GUEST$]
   check $CHECK$ to see if r is in the list of rooms to be checked [read $CHECK$]
   update $CHECK$ to delete the entry associated with r [write $CHECK$]

Consider the partial history $H =< S_{11}, S_{21}, S_{12} >$. S_{11}, S_{12} represents the two sections of the $Reserve$ transaction and S_{21} represents the low section of the $Cancel$ transaction. Suppose the room to be assigned to the guest is already in table $CHECK$. Step S_{22}, the high section of $Cancel$, cannot execute because its precondition, r is not in list of rooms to be checked, is not satisfied. $Check$ cannot be executed because the precondition of $Check$, r is not assigned to any guest, is not satisfied. No matter how many other transactions are initiated and executed, the preconditions of these sections will never be satisfied, and hence it is not possible to complete the partial history H. □

We define the notion of semantic atomicity to exclude the type of possibility illustrated in the previous example.

Definition 14 [Semantic Atomicity] Every partial multilevel semantic history H is a prefix of some complete multilevel semantic history.¶

The semantic atomicity guarantees that it is possible to complete any partial multilevel semantic history. Other transactions may have to be initiated to complete the history. The transactions that have to be initiated in order to complete a partial semantic history may be initiated by the system or by the user. When a transaction is being initiated by the system, we must ensure that no low section is being triggered to complete a high section of any transaction.

After defining the necessary properties of a multilevel semantic history we now proceed to define what it means for a multilevel semantic history to be correct.

Definition 15 [Correct Multilevel Semantic History] A multilevel semantic history is *correct* if it has the following three properties

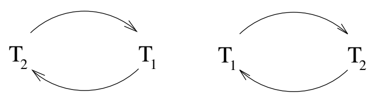
1. consistent execution property,
2. sensitive transaction isolation property, and
3. semantic atomicity property.

We finish by giving two examples of applications having semantic atomicity. The first one illustrates that we get more concurrency with semantic correctness than with serializability.

Example 9 [Additional Concurrency From Semantic Correctness] In this paper we have formalized the correctness of semantic histories. Semantic histories are sectionwise serial histories in which sections of different transactions are executed serially. There is a straightforward extension to the case of concurrent execution, where any multilevel history that is conflict equivalent [4] to a correct multilevel semantic history is acceptable.

Consider a $Reserve$ transaction T_1 and a $Cancel$ transaction T_2 of example 3. S_{11}, S_{12} are the two sections of the $Reserve$ transaction T_1; S_{21}, S_{22} are the two sections of the $Cancel$ transaction. The $Reserve$ transaction T_1 reserves

¶Garcia-Molina [9] has a slightly different definition of semantic atomicity, in that he allows for compensating steps. Compensation is problematic from the security perspective, so we omit it here.



(a) SG(H_4) (b) SG(H_6)

Figure 2. Serialization Graph For Example 9

a room for $John$. T_2 is a $Cancel$ transaction that cancels the reservation of room 28. A history that is conflict equivalent to any of the following histories will be acceptable if semantic correctness is used as the correctness criteria:
$H_1 =< S_{11}, S_{12}, S_{21}, S_{22} >$
$H_2 =< S_{21}, S_{22}, S_{11}, S_{12} >$
$H_3 =< S_{21}, S_{11}, S_{22}, S_{12} >$
$H_4 =< S_{21}, S_{11}, S_{12}, S_{22} >$
$H_5 =< S_{11}, S_{21}, S_{12}, S_{22} >$
$H_6 =< S_{11}, S_{21}, S_{22}, S_{12} >$
If traditional serializability is used as the correctness criteria, any of the histories H_1, H_2, H_3, or H_5 is acceptable.

The serialization graph of histories H_4 and H_6 (shown in figure 2) contain cycles, and are therefore unacceptable if serializability is used as the correctness criterion. Nonetheless, from a semantic perspective, the histories H_4 and H_6 are correct in that they satisfy the three properties. □

The next example helps to illustrate that isolation atomicity is not required for semantic atomicity. In other words, it is possible to interleave a transaction to establish the precondition for a high section.

Example 10 [Semantic Atomicity Without Isolation Atomicity] The problem in example 8 arose because the $Reserve$ transaction was incorrectly specified. A precondition was omitted from the second section of the $Reserve$ transaction which checks whether the room assigned to the guest is not in the list to be checked. The correct $Reserve$ transaction is given below.

1. check $STATUS$ to see if any room r has status $Available$ [read $STATUS$]
   update $STATUS$ by changing the status of r to $Unavailable$ [write $STATUS$]

2. check $CHECK$ to see if r is not in the list of rooms to be checked [read $CHECK$]
   check $GUEST$ to see if room r is not assigned to any guest [read $GUEST$]
   update $GUEST$ to add the entry corresponding to r and g [write $GUEST$]

The other transactions $Cancel$ and $Check$ are not changed. Note that the transactions $Reserve$ and $Cancel$ do not have the isolation atomicity property. Nevertheless this example enjoys semantic atomicity, even though example 8 lacks semantic atomicity. The sequence of sections $< S_{11}, S_{21}, S_{12} >$ in example 8 took the database to a deadlock state (room r assigned to a guest, and room r in the list of rooms to be checked) in which no sections of transactions (operating on the same room) could be executed. In this example we modified the $Reserve$ transaction by adding an extra precondition, which prevents the database from reaching this state.

To prove that the application has semantic atomicity we need to show that if any transaction T_i has completed section S_{ij}, it should be possible to execute a sequence of sections S_{mn},S_{pq},. . .,S_{rs} (the length of such a sequence may be 0 or more) , and eventually satisfy the precondition of the subsequent section $S_{i(j+1)}$ of T_i. In other words, if S_{ij} has been executed, we need to show that (i) the database is in a state in which the sequence of sections S_{mn},S_{pq},. . .,S_{rs} can be executed and (ii) the postcondition of executing the sequence should establish the precondition of $S_{i(j+1)}$.

The example application will have semantic atomicity if for any $Reserve$ or $Cancel$ transaction that has completed Section 1, it will be eventually possible to complete Section 2. Consider a $Reserve$ transaction that has completed Section 1 but cannot execute Section 2 because the precondition of Section 2 is not satisfied. When the precondition of Section 2 of $Reserve$ is not satisfied either (i) the precondition of $Check$ or (ii) the precondition of Section 2 of $Cancel$ is satisfied. In case (i), the postcondition of $Check$ establishes the precondition of Section 2 of $Reserve$. In case (ii), the postcondition of Section 2 of $Cancel$ establishes the precondition of $Check$, whose postcondition in turn establishes the precondition of Section 2 of $Reserve$. Thus the $Reserve$ transaction can always be completed. Similarly we can show that if the $Cancel$ transaction has executed only the first section, it can eventually be completed. The application thus has semantic atomicity. □

6. Discussion and Conclusion

Our contribution in this paper is the development of a semantic notion of atomicity suitable for multilevel secure transactions. The specifications for the transactions in a given application are statically analyzed to determine whether they satisfy the consistent execution property, the sensitive transaction isolation property and the semantic atomicity property. If these properties are satisfied, the developer is assured that for all complete multilevel semantic histories, the final state is a consistent state, either all or none of the sections of multilevel transactions are present in the history, and that outputs reflect a consistent database state. Further advantages of the semantic approach are that isolation atomicity – which is required for atomicity based on conflict serializability – is not required, that more concurrency among sections is allowed, and that the underlying mechanism need not preserve serialization orders between security levels.

Not all applications can be implemented by our model. The satellite report application [6, 7] is one such example. Example 5 shows that the satellite application lacks the consistent execution property. If a developer wishes to implement this application with our model, then the semantics of the application must be revised until the necessary properties are satisfied. Of course, the given semantics can be assured if serializability is the correctness criterion [7].

When the hotel example was introduced, we asserted that its formal structure was equivalent to a military-oriented example. Consider a mission database that maintains information about asset (troops, aircraft, etc.) to threat assignments. The information of whether an asset is busy or not is naturally classified at a lower level than the information of the precise threat to which an asset has been assigned. Consider two transactions in the mission database: $Respond$, in which an idle asset is assigned to a resource, and $Cancel$, in which an asset assignment to some threat is canceled. The mission example has exactly the same formal structure as the hotel example.

The analyses of the examples given in this paper are informal, but a formalization of related analyses for non-secure applications is given using the Z specification language in [2]. The advantage of formalization is that greater assurance can be obtained that the necessary properties do indeed hold. For future work, we intend to automate to the extent possible the process of generating and discharging the proof obligations for a given application. Industrial-level tool support for such an endeavor is essential, and we are investigating the use of the B Method [1] to this end.

References

[1] J. R. Abrial. *B-Technology Technical Overview*. B-Core(UK) Ltd., The Oxford Science Park, Oxford, UK, 1993.

[2] P. Ammann, S. Jajodia, and I. Ray. Using formal methods to reason about semantics-based decomposition of transactions. In *Proceedings of the Twenty-First International Conference On Very Large Databases*, pages 218–227, Zurich, Switzerland, September 1995.

[3] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corporation, Bedford, MA, July 1975.

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

[5] B. T. Blaustein, S. Jajodia, C. D. McCollum, and L. Notargiacomo. A model of atomicity for multilevel transactions.

In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 120–134, Oakland, CA, May 1993.

[6] O. Costich and S. Jajodia. Maintaining multilevel transaction atomicity in multilevel secure database systems with kernelized architecture. In B. Thuraisingham and C. Landwehr, editors, *Database Security VI: Status and Prospects*, pages 249–265. North-Holland, Amsterdam, 1993.

[7] O. Costich and J. McDermott. A multilevel transaction problem for multilevel secure database system and its solution for the replicated architecture. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 192–203, Oakland, CA, May 1992.

[8] A. A. Farrag and M. T. Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[9] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[10] T. F. Keefe and W. T. Tsai. Multilevel concurrency control for multilevel secure database systems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 369–383, Oakland, CA, May 1990.

[11] H. F. Korth and G. Speegle. Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model. *ACM Transactions on Database Systems*, 19(3):492–535, September 1994.

[12] K. P. Smith, B. T. Blaustein, S. Jajodia, and L. Notargiacomo. Correctness criteria for multilevel secure transactions. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), Feb. 1996.