A Semantic-Based Transaction Processing Model for Multilevel Transactions[*]

Indrakshi Ray[†] Paul Ammann[‡] Sushil Jajodia[†]

Abstract

Multilevel transactions have been proposed for multilevel secure databases; in contrast to most proposals, such transactions allow users to read and write across multiple security levels. The security requirement that no high level operation influence a low level operation often conflicts with the atomicity requirement of the standard transaction processing model. In particular, others have shown that no concurrency control algorithm based on the standard transaction processing model can guarantee both atomicity and security. This conflict motivates us to propose an alternative semantic-based transaction processing model for multilevel transactions. Our model uses the semantics of the application to analyze an application and reason about its behavior. Our notion of correctness is based on semantic correctness instead of serializability as in the standard transaction processing model. Semantic correctness ensures that database consistency is maintained, transactions output consistent data, and all partially executed transactions complete. We show how an example application can be analyzed to assure semantic correctness and how this analysis can be automated. We also propose a simple timestamp-based multiversion concurrency control algorithm for transaction processing on a kernelized architecture. The advantages of our model over the standard transaction processing model are that atomicity can be assessed, and for some applications ensured via off line analysis, more concurrency is achieved, lesser synchronization between security levels is required, and a larger class of multilevel transactions can be processed.

1 Introduction

Most of the research in 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 objects that are at its level. The need for multilevel transactions – transactions that can read and write objects at different security levels – arises in many applications. The standard transaction processing model [BHG87] based on serializability is not appropriate for multilevel transactions – executing multilevel transactions using mechanisms based on the standard transaction processing model may result in illegal information flow. Although the serializability model has been adapted for multilevel transactions [CM92, CJ93] atomicity cannot be ensured. In this work, we propose an alternative semantic-based transaction processing model which is appropriate for multilevel transactions. The chief advantage is that

[*] An earlier version of this paper appeared in IEEE Symposium on Security and Privacy, pages 74–84, Oakland, CA, May 1996.

[†] The work of Sushil Jajodia and Indrakshi Ray was partially supported by National Security Agency under grants MDA904–96–1–0103 and MDA904–96–1–0104 and by US Air Force/Rome Labs under grant F30602–97–1–0139. The work of Indrakshi Ray was also partially supported by a George Mason University Fellowship Award.

[‡] The work of Paul Ammann was partially supported by US Air Force/Rome Labs under grants F30602–97–1–0139.

atomicity can be assessed in advance via off line analysis. In addition, our model achieves greater concurrency, requires lesser synchronization between security levels, and can process a larger class of multilevel transactions than the algorithms based on the standard transaction processing model.

We first motivate the requirement for multilevel transactions. By definition, a single-level transaction prevents update operations at different levels from being grouped as an atomic transaction. Users, on the other hand, may need to execute a number of operations at different security levels as an atomic transaction. An example will help to illustrate this point. Suppose there is a mission database that maintains threat/resource information in a military environment. A resource is in a busy or idle state depending on whether or not it has been assigned to some threat. The information of whether a resource is busy or not is classified at a lower level (secret) than the information of the resource to threat assignment (top secret). The transaction *Respond* is responsible for assigning a resource to some threat. The *Respond* transaction performs two operations: (i) it picks an idle resource and changes its state to busy, and (ii) assigns the resource picked in operation (i) to the threat. The user submitting this transaction must be cleared to the top secret level. Note that *Respond* updates at multiple security levels; hence it cannot be modeled as one single-level transaction. To execute *Respond* using single-level transactions, the user must first log-on at the level secret and execute operation (i) and subsequently log-on at level top secret and execute operation (ii). Executing *Respond* as two single-level transactions may be undesirable for two reasons. First, it forces the user to manage the scheduling of the single-level transactions. Even worse, interleavings with other transactions may produce incorrect results. Consequently, multilevel transactions [BJMN93, CJ93, CM92, SBJN96] 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. To minimize the size of trusted code, a multilevel transaction is decomposed into single-level sections; a section contains all operations at the same security level.

Almost all the work in the area of multilevel transaction is based on the standard transaction processing model [BHG87]. The latter requires transactions to satisfy the atomicity, consistency, and isolation properties.[§] Consequently, multilevel transactions based on the standard transaction processing model are also required to satisfy these three properties. In addition, multilevel transactions must also satisfy the security property [SBJN96], which ensures that executing a multilevel transaction causes no illegal information flow across security levels. The atomicity requirement often conflicts with the security requirement of a multilevel transaction: a high section of a transaction may be unable to complete due to violations of the integrity constraints, and a rollback of low sections can be exploited to implement a covert channel. Smith et al. [SBJN96] prove that it is impossible to have concurrency control algorithms based on the standard transaction processing model that ensures the simultaneous satisfaction of the atomicity and security properties. This motivates us to propose an alternative semantic-based model for processing multilevel transactions.

Our model uses the semantics of the transaction to reason about correct and incorrect behavior of the application. Like other researchers [CM92, CJ93], we decompose a multilevel transaction into single-level sections. We execute each section atomically. Decomposing transactions into atomic sections results in the loss of the atomicity, consistency and isolation properties. To remedy this loss, we propose a set of replacement properties which we call the *semantic atomicity property*, the *consistent execution property* and the *sensitive transaction isolation property*. In the case of single level transactions, these properties reduce to the traditional properties. The new properties are defined in terms of *semantic histories* which are necessary to reason about correct and incorrect interleavings of sections of transactions. The semantic atomicity property ensures that all partially executed transactions complete, or, in other words, either all or none of the sections of a transaction

[§] The fourth standard property, durability, is exactly the same for secure databases as for traditional databases and so we do not mention it further.

appear in a history. The consistent execution property ensures that if a complete semantic history executes in a consistent state, the final state is also consistent. The sensitive transaction isolation property ensures that no inconsistencies are displayed to the user. These properties are more relaxed than their counterparts in the standard transaction processing model, and form the basis of semantic correctness. In addition to the replacement properties we specify two more properties, namely the *composition property* and the *isolation atomicity property.* The composition property ensures that a decomposition correctly models the original transaction. The isolation atomicity property ensures that if the lowest level section of a transaction has been executed, then it will be possible to execute all the other sections of the transaction when the transaction is executed in isolation. The isolation atomicity property is necessary if transactions are to be executed by algorithms based on the standard transaction processing model [CM92, CJ93]; this property, however, is optional in our semantic-based model.

The properties generate a set of proof obligations for a given application. The proof obligations must be successfully discharged to get assurance of the properties. As a feasibility study, we show how to get assurance of these properties for a specific application by enumerating and discharging the associated proof obligations using the Z specification language [Spi92]. For large complex applications, it is desirable to automate as much as possible the process of discharging the proof obligations. One solution is to use a theorem prover to automatically discharge the proof obligations. However, theorem proving is tedious, time consuming, difficult, and requires substantial expertise. This motivated us to adopt an alternate software verification technique known as model checking for our analysis. We show how the SMV model checker [McM92, CGL94] can be used, in part, to automatically discharge the proof obligations.

It is worth noting that the various properties are *correctness* properties rather than *security* properties. In other words, although an erroneous proof may result in an application that does not behave as intended with respect to functionality, the erroneous proof does not result in a security breach.

Once all the properties have been proved, the application must be executed by some concurrency control algorithm. Towards this end, we develop a simple timestamp based mechanism on a kernelized architecture for concurrently executing the decomposed multilevel transactions. The advantages of our model over the standard transaction processing model are that our model provides more concurrency, requires lesser synchronization between security levels and can process multilevel transactions which do not have isolation atomicity.

1.1 Related Work

The earliest works on multilevel transactions were done by Costich, Jajodia and McDermott [CM92, CJ93]; the authors present concurrency control algorithms based on kernelized [CJ93] and replicated [CM92] architectures which produce one-copy serializable histories of multilevel transactions. The algorithm proposed by Costich and McDermott [CM92] has a restriction: it can only execute transactions in which no low data item is written after accessing a high data item. This algorithm also requires the operations in a transaction to be ordered according to the security lattice structure. This requirement is obviated in the algorithm proposed by Costich and Jajodia [CJ93] and it accepts multilevel transactions where operations are not necessarily arranged in the order of security levels. This algorithm [CJ93] saves all the versions written by a transaction, and by using an indexing technique records the version of x that must be given to a read x operation. This technique always gives the correct version of x although the execution order may be different from that specified in the transaction. In both these works, the authors make an assumption – if one section of a transaction is successfully executed then it will be possible to execute all the other sections successfully.

This assumption makes the algorithms proposed by Costich and others [CM92, CJ93] suitable for only a limited class of multilevel transactions – transactions satisfying isolation atomicity property [AJR96].

Blaustein et al. [BJMN93] discuss the problem of ensuring atomicity for multilevel transactions and propose three degrees of atomicity for multilevel transactions – ML atomicity, L atomicity and complete atomicity. ML atomicity requires that if a section commits, then all sections at dominated levels must also commit. L atomicity requires that if a section commits, then all sections dominated by security level L must also commit. Complete atomicity requires that if a section commits then all other sections must also commit; complete atomicity corresponds to the traditional atomicity [GR93]. The authors also propose the notion of an execution graph which determines if the atomicity requirement of a multilevel transaction can be satisfied. The authors give 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 section 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 guarantee complete atomicity of multilevel transactions.

A formal treatment of multilevel transactions is presented by Smith et al. [SBJN96]. The authors define four correctness properties of multilevel transactions. These are *atomicity*, *consistency*, *isolation* and *security*. Atomicity (A correctness) requires either all or none of the operations in a transaction to commit. Consistency (C correctness) requires that the order of conflicting operations in a transaction be preserved in the schedule. Isolation (I Correctness) requires that the concurrent execution of a set of transactions be equivalent to the serial execution of the set of transactions. Security (S Correctness) requires that in a schedule no operations at a dominated level is influenced by any operation at a dominating level. The authors argue why it is impossible to give algorithms which guarantee the satisfaction of all four correctness properties. More specifically, it is impossible to give algorithms which guarantee both A correctness and S correctness. The authors also define partial correctness criteria based on relaxing the properties. The relaxed atomicity criterion, referred to as A^- correctness, corresponds to ML atomicity [BJMN93]. S^- correctness, the relaxed security criterion, is like S correctness, except that it allows timing channels. I^- correctness, the relaxed isolation property, achieves only degree 2 isolation [GR93].

The authors [SBJN96] give three algorithms : (i) Low-Ready-Wait, (ii) Low-First with Multiversion Timestamp Order, and (iii) Low-First with Hybrid Multiversioning. The Low-Ready-Wait algorithm is based on two phase locking; the sections are executed in the ascending order and are committed in the descending order of security levels. This algorithm ensures atomicity, consistency, isolation and relaxed security. In the Low-First with Multiversion Timestamp Ordering the sections are executed and committed in the ascending order of security levels. A unique timestamp is assigned to the transaction; sections inherit the timestamps of the transaction. If a section reads from a strictly dominated section of another transaction with an earlier timestamp, then it must wait for the completion of these strictly dominated sections. This algorithm ensures ML-atomicity, consistency, isolation and security. In Low-First with Hybrid Multiversioning the sections are executed and committed in the ascending order of security levels. Unlike Low-First with Multiversion Timestamp Ordering algorithm, sections are assigned unique timestamps. For reading and writing data items at its own level, a section follows the locking rules. For read downs, the timestamp ordering rules determine the version of the data item that will be read. This algorithm ensures ML-atomicity, consistency, security and degree 2 isolation.

The shortcomings of the standard transaction processing model and the need for alternative semantic-based transaction processing approaches is well researched in the context of long duration transactions. In these works [AAS93, AJR97, BL96, FÖ89, GM83, GMS87, KS94], the authors have introduced the notions of transaction decomposition, transaction steps, compensating steps, allowed versus prohibited interleavings of steps, and implementations in locking environments. The correctness of executions is based on semantic correctness and not on serializability. The goal of these works is to provide better performance than that provided by the standard transaction processing model.

Semantic-based transaction processing in the multilevel secure domain has been proposed by Ammann et al. [AJR96]. In this work the semantics of the application is used to statically analyze an application and determine its behavior. Correctness of the application was assessed in terms of satisfaction of the necessary properties. However in this work [AJR96] the ideas were informally presented. Our current work extends the work [AJR96] in the following ways: formalization of the properties, generating and discharging the proof obligations for a typical application, automated verification of the properties for the application using model checking approach, developing a notion of correctness for concurrent execution of a set of multilevel transactions, and giving a concurrency control algorithm based on a kernelized architecture.

1.2 Organization of the Paper

The remainder of the paper is organized as follows. Section 2 presents a motivating example – the mission database. Section 3 discusses the decomposition of multilevel transactions into sections; this section also discusses how to arrange the sections according to the security poset (partially ordered set) structure which is necessary for reasons of security. Section 4 discusses the details of our model. Section 5 presents the notion of correctness for concurrent execution of multilevel transactions. Section 6 describes a concurrency control algorithm based on a kernelized architecture. Section 7 concludes the paper. Appendix A illustrates how the the properties can be proved by hand for the mission example. Appendix B describes automated verification of the properties for the mission example with the SMV model checker [McM92, CGL94].

2 A Semantic View of Multilevel Transactions

We consider a security structure that is a partial order, $(\mathbf{C}, <)$. $\mathbf{C}$ is a set of security levels or classes, and $<$ is the dominance relation between classes. If $C_1 \leq C_2$, C_2 is said to dominate C_1 and C_1 is said to be dominated by C_2. If $C_1 < C_2$, C_2 is said to strictly dominate C_1 and C_1 is said to be strictly dominated by C_2. Two classes C_1 and C_2 are said to be incomparable if neither $C_1 \leq C_2$ nor $C_2 \leq C_1$.

A database is composed of a collection of objects where each object is associated with a single security level. At any given time, the *state* is determined by the values of the objects in the database. A change in the value of a database object changes the state. A database also has some predicates defined on the objects. These predicates are known as *invariants* or *integrity constraints*. A database state is said to be *consistent* if the values of the objects satisfy the given integrity constraints; otherwise the database is said to be 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.

$\mathbb{N}$	Set of Natural Numbers
$\mathbb{P}\,A$	Powerset of Set A
$\mathrm{bag}\,A$	Bag or MultiSet A
$\#A$	Cardinality of Set A
$\cup$	Set Union
$\setminus$	Set Difference (Also schema 'hiding')
$\uplus$	Bag (Multiset) Union
$\uminus$	Bag (Multiset) Difference
$A \fatsemi B$	Forward Composition of A with B
$x \mapsto y$	Ordered Pair $(x,\ y)$
$A \pfun B$	Partial Function from A to B
$A \pinj B$	Partial Injective Function from A to B
$B \ndres A$	Relation A with Set B Removed from Domain
$A \rhd B$	Relation A with Range Restricted to Set B
$\mathrm{dom}\,A$	Domain of Relation A
$\mathrm{ran}\,A$	Range of Relation A
$A \oplus B$	Function A Overridden with Function B
$x?$	Variable $x?$ is an Input
$x!$	Variable $x!$ is an Output
x	State Variable x before an Operation
x'	State Variable x' after an Operation
$\llbracket x \rrbracket$	Bag containing x
ΔA	Before and After State of Schema A
ΞA	ΔA with No Change to State

Table 1: Relevant Z Notation

A section of level C can view only those integrity constraints which involve objects at level C or below. Consequently, only the highest section, if present, 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 ensure that if a multilevel transaction executes on a consistent state, the result is again a consistent state. Postconditions of a multilevel transaction can update objects at different security levels; this is in contrast to single-level transactions where postconditions can update objects at only one security level.

2.1 An Example Illustrating Our View

In this paper we adopt the Z specification language for formalizing our ideas. Z is based on set theory, first order predicate logic, and a schema calculus to organize large specifications. Knowledge of Z is helpful, but not required, for reading this paper, since we narrate the formal specifications in English. Table 1 briefly explains the Z notation used in our examples. Other specification and analysis conventions peculiar to Z are explained as the need arises.

We illustrate our view with an example of a mission database. A Z specification appears in figure 1. The mission database has a set of objects, three integrity constraints on these objects, and three

$$[\mathit{Threat}, \mathit{Resource}]$$
$$\mathit{Status} ::= \mathit{Idle} \mid \mathit{Busy}$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Mission}} \\ \hline
\mathit{ASSIGN} : \mathit{Resource} \mathrel{⇸} \mathit{Threat} \\
\mathit{STATUS} : \mathit{Resource} \mathrel{⇸} \mathit{Status} \\ \hline
\operatorname{dom}(\mathit{STATUS} \rhd \{\mathit{Busy}\}) = \operatorname{dom} \mathit{ASSIGN} \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Respond}} \\ \hline
\Delta \mathit{Mission} \\
t? : \mathit{Threat} \\
r? : \mathit{Resource} \\ \hline
\mathit{STATUS}(r?) = \mathit{Idle} \\
\mathit{STATUS}' = \mathit{STATUS} \oplus \{r? \mapsto \mathit{Busy}\} \\
\mathit{ASSIGN}' = \mathit{ASSIGN} \cup \{r? \mapsto t?\} \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Cancel}} \\ \hline
\Delta \mathit{Mission} \\
r? : \mathit{Resource} \\ \hline
r? \in \operatorname{dom} \mathit{ASSIGN} \\
\mathit{STATUS}' = \mathit{STATUS} \oplus \{r? \mapsto \mathit{Idle}\} \\
\mathit{ASSIGN}' = \{r?\} \mathbin{⩤} \mathit{ASSIGN} \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Report}} \\ \hline
\Xi \mathit{Mission} \\
\mathit{currentstatus}! : \mathit{Resource} \mathrel{⇸} \mathit{Status} \\
\mathit{currentassignments}! : \mathit{Resource} \mathrel{⇸} \mathit{Threat} \\ \hline
\mathit{currentstatus}! = \mathit{STATUS} \\
\mathit{currentassignments}! = \mathit{ASSIGN} \\ \hline
\end{array}
$$

Figure 1: Initial Specification of the Mission Database

types of transactions, which we identify and explain below. The specification assumes two types, *Threat* and *Resource*, which enumerate all possible resources and all possible threats, respectively.

In Z states, as well as operations, are described with a two-dimensional notation called a *schema*, in which declarations for the objects appear in the top part and constraints on the objects appear in the bottom part. The objects in the mission database are listed in the schema *Mission*, which defines the state of the mission. The object *STATUS* is a partial function that records the status of each resource which may be *Busy* or *Idle*. The information whether a resource is busy or not is classified as secret and so the object *STATUS* has a security level secret. The object *ASSIGN* is a partial function that relates the resources with status *Busy* and the threats to which these resources are assigned. The information of the actual resource to threat assignment is classified at a higher level, namely top secret. Thus the object *ASSIGN* has a security level top secret. Note that the partial functions representing the objects *STATUS* and *ASSIGN* implicitly captures the following integrity constraints: (i) each resource can be either busy or idle, and (ii) each resource can be assigned to at most one threat. An additional integrity constraint on the objects in mission database appear in the bottom part of schema *Mission*: $\operatorname{dom}(\mathit{STATUS} \rhd \{\mathit{Busy}\}) = \operatorname{dom} \mathit{ASSIGN}$ – this states that the set of resources with status busy is exactly the set of resources assigned to threats. Note that this integrity constraint involves objects at different security levels: secret and top secret.

The three types of transactions in the mission database are *Respond*, *Cancel* and *Report* as shown in figure 1. *Respond* takes as input a resource $r?$ and a threat $t?$. *Respond* has a precondition that some resource $r?$ must be *Idle*. *Respond* has a postcondition that the status of $r?$ is changed to *Busy*, and the ordered pair $r? \mapsto t?$ is added to the function *ASSIGN*. *Cancel* cancels an assignment of a resource to a threat. *Cancel* has a precondition that $r?$ must be assigned to some threat (that is, $r?$ must be in the domain of *ASSIGN*) and a postcondition that $r?$ is removed from the domain of the function *ASSIGN*, and status of $r?$ is changed to *Idle*. Note that the postconditions in *Respond* and *Cancel* update objects at different security levels; hence they cannot be modeled using single-level transactions. *Report* has no preconditions and prints the objects *STATUS* and *ASSIGN* as outputs.

3 Decomposition of Multilevel Transactions

In the previous section we showed how multilevel transactions can be specified using preconditions and postconditions. Now a precondition check evaluates to reading one or more database objects and performing some check. A postcondition may involve updating an object; this update may depend on the values of the other objects in the database. Preconditions involve reading objects, and postconditions may involve reading and writing objects. Thus we define multilevel transactions as follows:

Definition 1 [**Multilevel Transaction**] A multilevel transaction T_i is a set of read and write operations in which conflicting operations [BHG87] are related by the partial order $\ll_i$.

A read or write operation in a multilevel transaction is associated with a security level. The security level of a write operation must be equal to the level of the object it writes. The security level of a read operation must dominate the level of the object it reads. A range of security levels is associated with a multilevel transaction. Each operation in the multilevel transaction must lie within this range. The user's clearance level at a particular session must dominate the range of the multilevel transaction submitted by the user in that session.

To prevent direct violations of the usual mandatory access control policy [BL75] by a multilevel transaction, we follow other authors [CM92, CJ93, SBJN96] and decompose each multilevel transaction into a set of *sections*, where each section is associated with a single security level. Each section is also specified with preconditions and postconditions. 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 [San93] :

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 2 [**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 related by the partial order $\ll_{ij}$,
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 poset structure (low to high order), as the following example illustrates. Consider three levels: L (Low) , M (Medium) , and H (High); two objects: x (L) , y (M); and one transaction: $w_L[x]\ r_H[x]\ r_H[y]\ r_M[x]\ w_M[y]$. In this example, the ordering of sections is L, H, M, and the medium section is delayed until the high section completes. The medium section can detect when the low section completes. The high section can modulate its execution time, thereby covertly passing information to the medium section. However given a transaction decomposed according to Definition 2 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 $\ll_i$.

Note that this problem has been solved by Costich and Jajodia [CJ93], and Smith et al. [SBJN96]. Our solution, which is based on that of Smith's [SBJN96], is presented next.

3.1 Ordering Sections in a Transaction

Before we show how to order the sections in a transaction according to the security poset structure, we need the notion of dependencies developed by Blaustein et al. [BJMN93]. 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. 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, $r_i[x] \ll_i w_i[x]$
2. 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, $w_i[x] \ll_i r_i[x]$

We adopt a caching technique [SBJN96] as the mechanism to correctly treat the H-L and L-H dependencies. The cache is local to the transaction; it is visible only to sections of a single transaction. The cache consists of a number of objects called cached objects. For all practical purposes cached objects are treated as database objects. Note that a cached object is accessed only by sections of a single transaction. In accordance with security requirements, a high section can read the cached objects written by the low section but not vice-versa. The cache is used by the transaction as long as it is active; once a transaction completes the cache associated with the transaction can be discarded.

The following algorithm shows how we treat the L-H and H-L dependencies using the cache. A section which writes a database object x writes the old value and the new value of x in the cache. In other words, two new cached objects, ox and nx, are created for each write x operation. From the database perspective, the objects x, ox, nx are distinct. The old value, new value of x are copied into ox, nx respectively. In the original transaction if there is a L-H dependency involving x, the read x operation in the high section is transformed into read nx. If there is a H-L dependency involving x, the read x operation in the high section is converted to a read ox operation and this read ox operation is moved after write x operation of the low section.

The caching technique converts the read operations on database objects to read operations on cached objects. 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 3 [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, $S_{ij} \prec_i S_{ik}$ 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 poset structure.

Definition 4 **[Canonical Form]** A transaction T_i is in *canonical form* if for all j, k, $S_{ij} \prec_i S_{ik}$ iff $L(S_{ij}) < L(S_{ik})$.

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 [BJMN93]; our result is to achieve semantic atomicity, where either all or none of the sections are executed.

The sections of the decomposed transactions for the mission database are shown in figure 2. The *Respond* transaction is decomposed into a secret section $R1$ and a top secret section $R2$. Similarly the *Cancel* transaction is decomposed into sections $C1$ and $C2$ where level of $C1$ is secret and that of $C2$ is top secret. For each of these transactions, the secret level section must be executed before the top secret level section is executed; thus *Respond* and *Cancel* transactions are in the canonical form. The *Report* transaction is composed of a single top secret section.

After a section has been executed, the original integrity constraints may no longer be satisfied. For this reason, we generalize the integrity constraints; these generalized constraints are satisfied before and after the execution of each section [AJR95]. The process of discovering appropriate generalized integrity constraints is outside the scope of this paper. For the seminal discussion of this topic in the context of concurrent execution of programs, see Owicki and Gries [OG76].

The generalized constraint for the mission database is given in the constraint part of the schema *DecomposedMission*. Notice the use of auxiliary variables, $R1ax$, $R2ax$, $C1ax$, $C2ax$, in the generalized constraint. The auxiliary variables are of type bag *Resource*. (A bag, also known as multi-set, is like a set except that the number of occurrences of each object in the bag is significant.) All these variables are initialized to the empty bag. A resource $r?$ whose status is changed to *Busy* in section $R1$ is included in the bag $R1ax$ in section $R1$. Finally when the resource is assigned to the threat in section $R2$, it is added to the bag $R2ax$. Thus the expression $\mathrm{dom}(R1ax \uplus R2ax)$ gives the resources whose status have been changed to *Busy* but which have not yet been assigned to threats, that is, resources which are in the process of being assigned. Similarly, the expression $\mathrm{dom}(C1ax \uplus C2ax)$ gives the resources with status *Idle* but which are still assigned to threats, that is, resources that are in the process of being cancelled. Note that the auxiliary variables are introduced for the purpose of analysis and can be omitted from the implementation. The *DecomposedMission* includes the schemas *MissionSecret* and *MissionTopSecret*. *MissionSecret* defines the secret level objects and the constraints defined on the secret level objects. Similarly, *MissionTopSecret* defines top secret level objects and constraints involving top secret level objects.

Note that additional preconditions are introduced in sections $R2$,$C1$, *Report*1. The precondition in $R2$ ensures that the integrity constraint, each resource can be assigned to at most one threat, is not violated. The precondition in $C1$ ensures that the assignment of a busy resource is canceled. The preconditions in *Report*1 ensure that no inconsistencies are displayed to the user by ensuring that there are no *Respond* transactions that have executed $R1$ but not $R2$ and that there are no *Cancel* transactions that have executed $C1$ but not $C2$.

After showing how transactions in an example can be decomposed, we are now ready to define our notion of correctness.

4 Semantic Correctness

In our model we use semantic correctness, rather than serializability, as the correctness criterion. In this section we develop the notion of semantic correctness by giving a set of properties which ensures that the application does not behave in an undesirable and unexpected way. The first two

$$\begin{array}{l}\mathit{MissionSecret}\\ \hline \mathit{STATUS} : \mathit{Resource} ⇸ \mathit{Status}\\ \mathit{R1ax}, \mathit{C1ax} : \mathrm{bag}\ \mathit{Resource}\\ \hline\end{array}$$

$$\begin{array}{l}\mathit{MissionTopSecret}\\ \hline \mathit{ASSIGN} : \mathit{Resource} ⇸ \mathit{Threat}\\ \mathit{R2ax}, \mathit{C2ax} : \mathrm{bag}\ \mathit{Resource}\\ \hline\end{array}$$

$$\begin{array}{l}\mathit{DecomposedMission}\\ \hline \mathit{MissionSecret}\\ \mathit{MissionTopSecret}\\ \hline \mathrm{dom}(\mathit{STATUS} \rhd \{\mathit{Busy}\}) \cup\\ \mathrm{dom}(\mathit{C1ax} \uplus \mathit{C2ax})\\ = \mathrm{dom}\ \mathit{ASSIGN} \cup \mathrm{dom}(\mathit{R1ax} \uplus \mathit{R2ax})\\ \hline\end{array}$$

$$\begin{array}{l}\mathit{R1}\\ \hline \Delta \mathit{DecomposedMission}\\ \Xi \mathit{MissionTopSecret}\\ r? : \mathit{Resource}\\ \hline \mathit{STATUS}(r?) = \mathit{Idle}\\ \mathit{STATUS}' = \mathit{STATUS} \oplus \{r? \mapsto \mathit{Busy}\}\\ \mathit{R1ax}' = \mathit{R1ax} \uplus \llbracket r? \rrbracket\\ \mathit{C1ax}' = \mathit{C1ax}\\ \hline\end{array}$$

$$\begin{array}{l}\mathit{C1}\\ \hline \Delta \mathit{DecomposedMission}\\ \Xi \mathit{MissionTopSecret}\\ r? : \mathit{Resource}\\ \hline \mathit{STATUS}(r?) = \mathit{Busy}\\ \mathit{STATUS}' = \mathit{STATUS} \oplus \{r? \mapsto \mathit{Idle}\}\\ \mathit{C1ax}' = \mathit{C1ax} \uplus \llbracket r? \rrbracket\\ \mathit{R1ax}' = \mathit{R1ax}\\ \hline\end{array}$$

$$\begin{array}{l}\mathit{R2}\\ \hline \Delta \mathit{DecomposedMission}\\ \Xi \mathit{MissionSecret}\\ t? : \mathit{Threat}\\ r? : \mathit{Resource}\\ \hline r? \notin \mathrm{dom}\ \mathit{ASSIGN}\\ \mathit{ASSIGN}' = \mathit{ASSIGN} \cup \{r? \mapsto t?\}\\ \mathit{R2ax}' = \mathit{R2ax} \uplus \llbracket r? \rrbracket\\ \mathit{C2ax}' = \mathit{C2ax}\\ \hline\end{array}$$

$$\begin{array}{l}\mathit{C2}\\ \hline \Delta \mathit{DecomposedMission}\\ \Xi \mathit{MissionSecret}\\ r? : \mathit{Resource}\\ \hline r? \in \mathrm{dom}\ \mathit{ASSIGN}\\ \mathit{ASSIGN}' = \{r?\} ⩤ \mathit{ASSIGN}\\ \mathit{R2ax}' = \mathit{R2ax}\\ \mathit{C2ax}' = \mathit{C2ax} \uplus \llbracket r? \rrbracket\\ \hline\end{array}$$

$$\begin{array}{l}\mathit{Report1}\\ \hline \Xi \mathit{DecomposedMission}\\ \mathit{currentstatus}! : \mathit{Resource} ⇸ \mathit{Status}\\ \mathit{currentassignments}! : \mathit{Resource} ⇸ \mathit{Threat}\\ \hline \mathit{R1ax} = \mathit{R2ax}\\ \mathit{C1ax} = \mathit{C2ax}\\ \mathit{currentstatus}! = \mathit{STATUS}\\ \mathit{currentassignments}! = \mathit{ASSIGN}\\ \hline\end{array}$$

Figure 2: Decomposed Mission Database

properties, namely, the composition property and the isolation atomicity property, describes the behavior of the transaction when it is executed in isolation and without interference from any other transaction. The last three properties, namely, semantic atomicity property, consistent execution property and sensitive transaction isolation property, consider the behavior of the transaction when the transaction is not executed in isolation, that is, when sections of one transaction are interleaved with those of another. Note that the last three properties substitute for the atomicity, consistency and isolation properties of the standard transaction processing model.

4.1 Composition Property

The composition property ensures that the transformation applied to the original transaction does not change the semantics of the transaction when considered by itself. In other words, the composition property formally verifies that executing the sections in an order that is consistent with the canonical ordering changes the database objects in the same way as changed by the original transaction.

[**Composition Property**] A multilevel transaction satisfies the composition property if for any given consistent state of the database, executing the sequence of sections on the consistent state is equivalent to executing the original transaction on the same state.

Formally the composition property is stated as follows:

Let I denote the original integrity constraints; T_i denote the original transaction and S_{i1},S_{i2}, ...,S_{in} denote the corresponding sections. The before state and after state of the auxiliary variables are represented by aux, aux' respectively.

A transaction T_i is said to have the composition property when

$$I \wedge (S_{i1} \mathbin{⨾} S_{i2} \ldots \mathbin{⨾} S_{in}) \setminus (aux, aux') \Leftrightarrow T_i$$

The above expression is written in Z. The left hand side represents the state formed by the sequential execution of the sections on a consistent state and the auxiliary variables suppressed from the resulting state. The right hand side represents state resulting from the execution of the original transaction T_i. The $\Leftrightarrow$ stands for equivalence. The proofs that the decomposed transactions in the mission database have the composition property are shown in Appendix A.

4.2 Isolation Atomicity Property

When sections of a transaction are executed sequentially on a consistent database state, it may not be possible to complete the transaction. This may happen if the precondition of some dominating section is not satisfied after the execution of some dominated section. The isolation atomicity property ensures that such a situation is avoided; if a transaction has been partially executed then it will be possible to complete the transaction.

[**Isolation Atomicity Property**] A multilevel transaction satisfies 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.

Formally the isolation atomicity property is stated as follows:

[¶] 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. Note that a transaction may not have a least section. For example, consider a transaction with three sections, two mutually incomparable and another that dominates the first two.

Let I denote the original integrity constraints; T_i denote the original transaction and S_{i1},S_{i2}, …,S_{in} denote the corresponding sections.

If S_{i1} is the least section, the transaction satisfies the isolation atomicity when

$$\mathrm{pre}\, S_{i1} \wedge I \Rightarrow \mathrm{pre}(S_{i1} \mathbin{⨾} S_{i2} \ldots S_{in})$$

The above expression is written in Z. The left hand side of the $\Rightarrow$ represents the state satisfying the precondition of the least section S_{i1} and the original integrity constraints. The right hand side denotes the state satisfying the precondition of the composition of the sections.

In cases where a transaction does not have a least section, the transaction is said to have isolation atomicity when

$$I \Rightarrow \mathrm{pre}(S_{i1} \mathbin{⨾} S_{i2} \ldots S_{in})$$

There are many examples of transactions not having isolation atomicity. For example, in the mission database, suppose we enforce a new requirement that each threat can be assigned to at most one resource. In other words, suppose we require *ASSIGN* to be injective. Then an additional precondition, namely, $t? \notin \mathrm{ran}\, ASSIGN$, must be added to the the second section of the *Respond* transaction. As a result, this modified *Respond* transaction will not have isolation atomicity. However, in the original mission example we have no such requirement and as it turns out all transactions have isolation atomicity – the proofs are given in Appendix A.

Isolation atomicity is a necessary property if transactions are to be executed by algorithms based on serializability. For such algorithms, if a transaction lacks isolation atomicity, then it may never complete, thus violating the atomicity requirement. To build on the example of the previous paragraph where *ASSIGN* is assumed to be injective, suppose that $R1$ executes successfully, but threat $t?$ is already associated with some resource in *ASSIGN*. Then $R2$ cannot complete. Note that in our model transactions are not executed in isolation and we are able to relax the isolation atomicity requirement. Completion of the transactions is guaranteed by the semantic atomicity property discussed in Section 4.4.

The next three properties we discuss are defined over histories. Before describing these properties we need the notion of histories.

4.3 Semantic Histories

So far we have considered what happens if transactions are executed in isolation, that is, without interference from other transactions. However sections of different transactions are allowed to interleave. 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 semantic history.

Definition 5 [**Sectionwise Serial One Version History**] A sectionwise serial one version history H defined over a set of multilevel 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.

Condition (1) ensures that every section of a transaction occurs at most once in a sectionwise serial one version history. Condition (2) ensures that the order of the sections in canonical form of a transaction is preserved in a sectionwise serial one version history. Condition (3) ensures that for every section S_{ik} in a sectionwise serial one version history, all the sections that precede S_{ik} in canonical form of T_i are present in the sectionwise serial one version history. Note that in a sectionwise serial one version history H if $S_{ij} \prec_H S_{kl}$, then all operations of S_{ij} must precede any operation of S_{kl}. Also note that $\prec_H$ is a total order.

A sectionwise serial one version history lists the sequence of steps in the history; it does not give any information about the state of the database in which the section is being executed. To reason about correct and incorrect interleavings, we need to know the states associated with a history. This motivates us to introduce the notion of *semantic* history which not only lists the sequence of sections forming the history, but also conveys information regarding the state of the database before and after the execution of each section in the history.

Definition 6 [**Partial Semantic History**] A *partial semantic* history H is a sectionwise serial one version history that is bound to

1. an initial state, and

2. the states resulting from the execution of each section in H.

Note that a semantic history H is associated with a sectionwise serial one version history. When we deal with the syntactic aspects of H, we refer to it as a sectionwise serial one version history; when we are interested in the semantic information – the states associated with the history H – we refer to it as a semantic history.

Definition 7 [**Complete Execution**] An execution of a transaction $T_i = S_{i1}, S_{i2}, \ldots, S_{in}$ in a sectionwise serial one version history H is a *complete* execution if all n sections of T_i appear in H.

Definition 8 [**Complete Semantic History**] A partial semantic history H over $\mathbf{T}$ is a *complete* semantic history if the execution of each T_i in $\mathbf{T}$ is *complete.*

In the following we identify three necessary properties which allowable semantic histories should possess.

4.4 Semantic Atomicity Property

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

[**Semantic Atomicity**] Every partial semantic history H is a prefix of some complete semantic history.[‖]

[‖] Garcia-Molina [GM83] 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.

Semantic atomicity guarantees that it is possible to complete any partial 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. When a transaction is being initiated by a user, the constraints are looser since users are trusted up to their clearance. For instance, reconsider the injective *ASSIGN* example discussed earlier at the point where $R1$ had executed but $R2$ was blocked because the desired threat $t?$ was already associated with some resource. A user could initiate a *Cancel* transaction to disassociate threat $t?$ from its current resource, at which point the $R2$ section of the *Reserve* transaction could finish.

4.5 Consistent Execution Property

An important property of databases is consistency. When transactions have been decomposed into sections, and sections of different transactions are allowed to interleave, the database must be restored to a consistent state after all the transactions complete. We capture this requirement as the consistent execution property.

[**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 consistent.

4.6 Sensitive Transaction Isolation Property

When a transaction has partially executed (that is, some but not all of its sections have committed) the database may be in an inconsistent state; sections of other transactions may be exposed to this inconsistency. In some cases this is problematic. For example, some transactions output data to users; these transactions are called *sensitive* transactions [GM83]. Sensitive transactions should not output any inconsistent data.

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

In our model, we ensure the sensitive transaction isolation property by construction. There are two aspects to such a construction. First, for each sensitive transaction, we compute the subset of the original integrity constraints, I, relevant to the calculation of any outputs. This subset of I must be implied by the preconditions of the section or sections that generate the outputs. Second, as pointed out by Rastogi, Korth, and Silberschatz [RKS95], if outputs are generated by multiple sections, interleavings between these sections must be controlled to ensure that outputs from later sections are consistent with outputs from earlier sections.

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

Definition 9 [**Correct Semantic History**] A semantic history is *correct* if it has the following three properties

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

The application developer must prove that all semantic histories generated from the application are correct. If the properties cannot be proved, the specification must be revised. Appendix A shows how the proof obligations corresponding to the above three properties can be discharged for the mission database. These proof obligations have been discharged using hand analysis. However for real world complex applications it is desirable to automate as much as possible the verification of the properties.

In theory one could use some theorem prover designed for the Z specification language to discharge the necessary proof obligations. We decided not to adopt this approach for two reasons. Firstly, Z does not have a trace-based semantics and does not allow for the specification of the necessary properties. Secondly, theorem proving is difficult, tedious and time consuming; consequently, even with the state-of-the-art theorem provers, such as the PVS [ORS92], only small applications can be verified. This motivated us to look at an alternative automated approach known as model checking to get assurance of the properties. Model checking has been used successfully to verify hardware circuits; only recently researchers [AG93, WVF96] are advocating this approach to check software.

Fundamental to model checking is its reliance on finite state machines. A model checker requires the system being verified be represented as a finite state machine, and the properties be represented as temporal formulae. The model checker then performs an exhaustive search of the state space to see whether the properties hold. Since software systems, in general, are infinite state machines, the model checker cannot directly verify software systems. To solve this problem, researchers [WVF96, Jac96] propose developing finite state abstractions of the software system which can be verified by model checkers. In Appendix B we describe how we develop a finite model of the mission application which we verify using the SMV model checker [McM92, CGL94].

Once we prove the necessary properties for a given application, we get the assurance that all semantic histories generated from the application are correct. Note that semantic histories are sectionwise serial one version histories in which sections of transactions are executed serially. The following section describes the notion of correctness when sections are executed concurrently.

5 Concurrent Executions

In the previous section we have outlined the correctness criterion for semantic histories. Note that in semantic histories, which are sectionwise serial one version histories, the sections are executed serially. To improve the throughput, sections of one transaction must be executed concurrently with those of another transaction. Our eventual aim is to get a concurrency control mechanism in which sections need not be executed atomically, but the read, write operations of one section can be interleaved with those of another. In this section we describe our notion of correctness in the face of concurrent execution of sections. From now on we focus on complete histories and use the word history to refer to a complete history. We begin by giving a definition of history.

Definition 10 [History] A history H defined over a set of multilevel transactions $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$, where each transaction T_i has been decomposed into i_n sections, is a partial order with ordering relation $\ll_H$ where:

1. $H = \cup_{i=1}^{m} \cup_{j=1}^{i_n} S_{ij}$;

2. $\ll_H \supseteq \cup_{i=1}^{m} \ll_{ij}$; and

3. for any two conflicting operations $p, q \in H$, either $p \ll_H q$ or $q \ll_H p$.

Condition (1) says that the execution represented by H involves precisely the operations of the sections of T_1, T_2, ..., T_m. Condition (2) says that the H honors all conflict orderings specified within each section. Condition (3) says that every pair of conflicting operations are ordered in H.

All the definitions of the histories given so far assume that there is only one version of a data item in the database. These histories represent the order of execution of operations as viewed by the user. However, if the underlying database is a multiversion database, the system's view of histories will be different from the user's view. In a multiversion database, for each data item x, we denote the versions of x by x_{im},x_{jr},..., where x_{im}, x_{jr} are the versions written by section S_{im}, S_{jr} respectively. For each read and write operation, the system must translate the operation into an equivalent operation on some version of the data item. Let h denote the translation function. For a read operation $r_{ij}[x]$, h determines the version of x to be read; that is, if $h(r_{ij}[x]) = x_{pq}$, then the value of x_{pq} will be returned by the read operation. For a write operation $w_{ij}[x]$, h determines the version of x that will be created; $h(w_{ij}[x]) = x_{ij}$ then it means the write operation creates the version x_{ij}. To represent the system's view of the execution of operations we define a multiversion history.

Definition 11 [**Multiversion History**] A multiversion history H defined over a set of multilevel transactions $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$, where each transaction T_i has been decomposed into i_n sections, is a partial order with ordering relation $\ll_H$ where:

1. $H = h(\cup_{i=1}^{m} \cup_{j=1}^{i_n} S_{ij})$;
2. for each S_{ij} and all operations p_{ij}, q_{ij} in S_{ij}, if $p_{ij} \ll_{ij} q_{ij}$, then $h(p_{ij}) \ll_H h(q_{ij})$;
3. if $h(r_{ij}[x]) = r_{ij}[x_{kl}]$, then $w_{kl}[x_{kl}] \ll_H r_{ij}[x_{kl}]$; and
4. if $w_{ij}[x] \ll_{ij} r_{ij}[x]$, then $h(r_{ij}[x]) = r_{ij}[x_{ij}]$.

Condition (1) states that the multiversion history contains the translations of the operations in each section. Condition (2) states that the multiversion history preserves all the operation orderings specified by the sections. Condition (3) states that a section cannot read a version before it has been created. Condition (4) states that if a section writes a data item and subsequently reads it, then it must read the version written by itself.

Our objective is to characterize multiversion histories that are equivalent to sectionwise serial one version histories. For this we need the notion of equivalence of histories; this definition is similar to the one given by Bernstein *et al.* [BHG87]. Before defining equivalence we need the notion of reads from relationship.

Definition 12 [S_{ij} **reads** x **from** S_{kl}] For a one version history, S_{ij} reads x from S_{kl} if $w_{kl}[x] \ll_H r_{ij}[x]$ and there is no $w_{mn}[x]$ such that $w_{kl}[x] \ll_H w_{mn}[x] \ll_H r_{ij}[x]$. For a multiversion history, S_{ij} reads x from S_{kl} if the operation $r_{ij}[x_{kl}]$ is present in the history.

Definition 13 [**Equivalence of Histories**] Two histories H and H' are said to be equivalent, if they are defined over the same sections and have the same reads from relationship. Two multiversion histories H_1 and H_2 are equivalent if they have the same operations.

Next we define what it means for a multiversion history to be sectionwise serial.

Definition 14 [**Sectionwise Serial Multiversion History**] A multiversion history is sectionwise serial if for every two sections S_{ij} and S_{kl} that appear in H, either all of S_{ij}'s operations precede all of S_{kl}'s or vice versa. When all of S_{ij}'s operation precede all of S_{kl}'s operation, S_{ij} is said to precede S_{kl}.

However not all sectionwise serial multiversion histories are equivalent to sectionwise serial one version histories; this is exactly the same issue as in ordinary multiversion databases [BHG87]. The following definition characterizes those sectionwise serial multiversion histories which are equivalent to sectionwise serial one version histories.

Definition 15 [**One-Copy Sectionwise Serial Multiversion History**] A sectionwise serial multiversion history H is one-copy sectionwise serial multiversion history if

1. for all ij, kl, and x, if S_{ij} reads x from S_{kl}, then either $ij = kl$ or S_{kl} is the last section preceding S_{ij} that writes into any version of x.
2. if S_{ij} precedes S_{im} in H, then S_{im} does not precede S_{ij} in the canonical form of T_i.

Finally we define a one-copy sectionwise serializable multiversion history in which sections may not be executed serially, but the effect of the history is the same as a one-copy sectionwise serial history.

Definition 16 [**One-Copy Sectionwise Serializable Multiversion History**] A one-copy sectionwise serializable multiversion history is one that is equivalent to a one-copy sectionwise serial multiversion history.

In the following section we describe an algorithm which generates one-copy sectionwise serializable histories.

6 Concurrency Control Algorithm

Our algorithm is based on the algorithm by Costich and Jajodia [CJ93] and it is for a kernelized architecture. The scheduler is divided into two parts: trusted global scheduler which controls the proper sequencing of sections of a transaction, and untrusted local schedulers at each security level which are responsible for scheduling the database operations of a section.

Global scheduler: The decomposed transactions are submitted to the global scheduler. The global scheduler is ready to dispatch a section S_{ij} to the local scheduler when all the sections of T_i at levels dominated by $L(S_{ij})$ have completed execution. When S_{ij} is ready to be dispatched, the global scheduler generates a unique timestamp for S_{ij}, denoted by $ts(S_{ij})$, and then submits S_{ij} to the local scheduler at level $L(S_{ij})$.

Local Scheduler: Our algorithm assumes that all the read sets and write sets of a section are known in advance. The local scheduler does not execute section S_{ij} until all sections, at dominated levels with earlier timestamps whose write sets have a non-null intersection with the read set of S_{ij}, have committed. The local scheduler then processes the read operations as follows: each $r_{ij}[x]$ is translated into $r_{ij}[x_{kl}]$ where x_{kl} is the version written by section S_{kl} and S_{kl} is the section with the highest timestamp not greater than $ts(S_{ij})$ which writes into any version of x. The local scheduler processes write operation as follows: each $w_{ij}[x]$ is translated into $w_{ij}[x_{ij}]$. The local scheduler after completing the section successfully or unsuccessfully informs the global scheduler.

6.1 Proof of Correctness

Theorem 1 A history H generated by the concurrency control algorithm is a one-copy sectionwise serializable history.

Proof: Apply the following transformation on the history H: arrange the operations to get a sectionwise serial history H_s. The order of sections in H_s must be the same as the timestamp ordering. Now we will show that H_s is a one-copy sectionwise serial history. For this we must show two things: (i) if there is an operation $r_{ij}[x_{kl}]$ in H_s, then $w_{kl}[x_{kl}]$ is the last operation preceding $r_{ij}[x_{kl}]$ that writes into any version of x, and (ii) H_s preserves the canonical ordering of the transaction. We prove (i) by contradiction. Suppose $w_{ij}[x_{kl}]$ is not the last operation preceding $r_{ij}[x_{kl}]$ to write into any version of x. In other words, assume there is another operation say $w_{mn}[x_{mn}]$ which is the last write operation preceding $r_{ij}[x_{kl}]$ that writes a version of x. This means that $ts(S_{kl}) < ts(S_{mn}) < ts(S_{ij})$. This is not possible because in the algorithm the scheduler translated $r_{ij}[x]$ into $r_{ij}[x_{kl}]$ implies that S_{kl} is the section with the highest timestamp not greater than $ts(S_{ij})$ that writes into any version of x. Thus we arrive at a contradiction. This means that if S_{ij} reads x from S_{kl}, S_{kl} is the last section preceding S_{ij} that writes any version of x – thus (i) is satisfied. For (ii) the proof is as follows. In the algorithm, the global scheduler assigns a unique timestamp to a section and dispatches the section to the local scheduler only after the dominated sections of the same transaction have committed. Hence in a transaction, a section at a dominating level has a higher timestamp than a section at the dominated level. So in H_s, the section at dominated level will appear before section at dominating level. Thus H_s preserves the canonical ordering of a transaction. Since H_s satisfies (i) and (ii), it is a one-copy sectionwise serial history. Since H and H_s are composed of the same operations they are equivalent; therefore H is a one-copy sectionwise serializable history. □

6.2 Informal Proof of Security

Having proved that our algorithm generates one-copy sectionwise serializable histories, we now proceed to prove informally that our algorithm is secure. For proving security we must show the following: (i) there is no direct illegal information flow from dominating to the dominated levels, and (ii) our algorithm does not introduce any covert channels.

The multilevel transactions are decomposed into single-level sections. Sections of a transaction are allowed to read objects at dominated levels and write objects at their own level. Thus we enforce the simple security and the restricted *-property of the Bell-Lapadula model [San93]; hence there is no direct information flow from the dominating level to the dominated level.

The proof that our algorithm does not introduce any covert channel proceeds in two parts: (a) no operation at the dominated level is delayed because of some operation at the dominating level, (b) no section at dominated level is aborted because of some section at dominating level. First we show that no operation in a dominated section is delayed by any operation in the dominating section. Let S_{ij} be the dominated section and S_{kl} be the dominating section. If an operation in S_{ij} is delayed because of some operation in S_{kl}, then S_{ij} and S_{kl} must be accessing some common data item where S_{ij} writes the data item and S_{kl} reads it. For $i = k$, the proof is trivial: according to the algorithm the dominating section of a transaction is initiated only after the dominated section commits - thus an operation in S_{kl} cannot delay one in S_{ij}. For $i \neq k$, the proof is as follows. Since S_{ij} and S_{kl} are distinct, either $ts(S_{ij}) < ts(S_{kl})$ or $ts(S_{kl}) < ts(S_{ij})$. If $ts(S_{ij}) < ts(S_{kl})$, the algorithm will not begin executing S_{kl} until S_{ij} has committed; thus an operation in S_{ij} cannot be delayed by one in S_{kl}. If $ts(S_{kl}) < ts(S_{ij})$, section S_{kl} will not access any version of any data item written by S_{ij} and no operation in S_{ij} will be delayed because of an operation in S_{kl}.

Next we show that our algorithm does not abort a dominated section because of some operation in the dominating section. Let S_{ij} be the dominated section and S_{kl} be the dominating section. For $i = k$, the proof is trivial, as the dominated sections of a transaction are committed before the initiation of the dominating sections. For $i \neq k$, we must show that S_{ij} is not aborted because of S_{kl}. For this let us consider the section aborts that may occur for concurrency control reasons. Note that in timestamp based algorithm if a section S_{ij} with a smaller timestamp tries to write a data item after another section S_{kl} with a larger timestamp has already read the data item, section S_{ij} will be aborted. In our algorithm this possibility is avoided by requiring sections with later timestamps to wait until all sections, at dominated levels whose write sets have a non-null intersection with the read set of the dominating section, have completed execution. In other words, in our algorithm if S_{kl} has a larger timestamp than S_{ij}, S_{kl} will not be initiated until S_{ij} completes and S_{ij} will not be aborted for concurrency control reasons.

7 Conclusion

Our contribution in this paper is the development of a semantic-based transaction processing model for processing multilevel transactions. We have provided the application developer the conceptual tools necessary to reason about systems in which transactions that ideally should be treated as atomic – for reasons of analysis – must instead be treated as a composition of sections – for reasons of security. The developer begins with a specification produced via standard formal methods, decomposes some transactions in the specification into single-level sections, and assesses the properties of the resulting system. The formal analysis at each step of this process provides assurance that the resulting system possesses the desired properties.

Some advantages of the semantic approach are that isolation atomicity – required for algorithms based on syntactic approach [CJ93, CM92] – is not required, that more concurrency among sections is allowed, and that the underlying mechanism need not preserve serialization orders between security levels. The cost of semantic concurrency control is that the set of transactions in an application must be analyzed off-line to ensure the set of properties. 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. For ensuring the properties, proof obligations must be generated and discharged. A significant challenge of our model is the difficulty of discharging the proof obligations. Towards this end, in Appendix B we have shown how the SMV model checker [McM92, CGL94] can be used for analyzing applications of interest. Applications in which the proof obligation associated with some necessary property cannot be successfully discharged must be revised. Revising the applications without changing their semantics may not be easy, but at least the developer is aware of the precise trade offs between atomicity, consistency, isolation and security.

References

[AAS93] D. Agrawal, A. El Abbadi, and A. K. Singh. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Transactions on Database Systems*, 18(3):460–486, September 1993.

[AG93] J. M. Atlee and J. D. Gannon. State-based model checking of event driven systems requirements. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[AJR95] P. Ammann, S. Jajodia, and I. Ray. Using formal methods to reason about semantics-based decomposition of transactions. In *Proceedings of the International Conference on Very Large Databases*, pages 218–227, Zurich, Switzerland, September 1995.

[AJR96] P. Ammann, S. Jajodia, and I. Ray. Ensuring atomicity of multilevel transactions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 74–84, Oakland, CA, May 1996.

[AJR97] P. Ammann, S. Jajodia, and I. Ray. Applying formal methods to semantic-based decomposition of transactions. *ACM Transactions on Database Systems*, 22(2):215–254, June 1997.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[BJMN93] B. T. Blaustein, S. Jajodia, C. D. McCollum, and L. Notargiacomo. A model of atomicity for multilevel transactions. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 120–134, Oakland, CA, May 1993.

[BL75] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, The MITRE Corporation, Bedford, MA, July 1975.

[BL96] A. Bernstein and P. Lewis. High performance transaction systems using transaction semantics. *Distributed and Parallel Databases*, 4(1):25–47, 1996.

[CGL94] E. M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency – Reflections and Perspectives.* Springer Verlag, 1994. Lecture Notes in Computer Science 803.

[CJ93] O. Costich and S. Jajodia. Maintaining multilevel transaction atomicity in multilevel secure database systems with kernelized architecture. In B.M. Thuraisingham and C.E. Landwehr, editors, *Database Security VI: Status and Prospects*, pages 249–265. North-Holland, Amsterdam, 1993.

[CM92] 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.

[FÖ89] 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.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 249–259, San Francisco, CA, 1987.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Francisco, CA, 1993.

[Jac96] D. Jackson. Niptick: A checkable specification language. In *Proceedings of the Workshop on Formal Methods in Software Practice*, San Diego, CA, January 1996.

[KS94] 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.

[McM92] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1992.

[OG76] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of the International Conference on Automated Deduction*, pages 748–752, Saratoga, NY, June 1992.

[RKS95] R. Rastogi, H. F. Korth, and A. Silberchatz. Exploiting transaction semantics in multidatabase systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 101–109, Vancouver, Canada, June 1995.

[San93] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, November 1993.

[SBJN96] 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):32–45, February 1996.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[WVF96] J. M. Wing and M. Vazari-Farahani. A case study in model checking software systems. Technical Report CMU-CS-96-124, Carnegie Mellon University, Pittsburgh, PA, April 1996. To appear in Science of Computer Programming.

Appendix A – Verification of the Mission Decomposition

In this appendix we show how the properties can be proved manually for the decomposition of the transactions in the mission database. Note that the specifications of the decomposed transactions are given in Figure 2.

Composition Property

Proof Obligations:

(a) $\mathit{Mission} \land (R1 \fatsemi R2) \setminus (R1ax, R1ax', C1ax, C1ax') \Leftrightarrow \mathit{Respond} \cdots$ (i)
(b) $\mathit{Mission} \land (C1 \fatsemi C2) \setminus (R1ax, R1ax', C1ax, C1ax') \Leftrightarrow \mathit{Cancel} \cdots$ (ii)

Proof of (a):

The left hand side of (i) evaluates to

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Mission} \land (R1 \fatsemi R2) \setminus (R1ax, R1ax', C1ax, C1ax')} \\
\Delta \mathit{Mission} \\
t? : \mathit{Threat} \\
r? : \mathit{Resource} \\
\hline
\mathit{STATUS}(r?) = \mathit{Idle} \\
\mathit{STATUS}' = \mathit{STATUS} \oplus \{r? \mapsto \mathit{Busy}\} \\
\mathit{ASSIGN}' = \mathit{ASSIGN} \cup \{r? \mapsto t?\} \\
\hline
\end{array}$$

The predicate part of the schema is equivalent to the *Respond* schema given in Figure 1; hence the two schemas are equivalent.

Using similar arguments we can prove (b).

Isolation Atomicity Property

Proof Obligations:

(a) $\mathrm{pre}\, R1 \land \mathit{Mission} \Leftrightarrow \mathrm{pre}(R1 \fatsemi R2) \cdots$ (i)
(b) $\mathrm{pre}\, C1 \land \mathit{Mission} \Leftrightarrow \mathrm{pre}(C1 \fatsemi C2) \cdots$ (ii)

Proof of (a):

The precondition schemas $\mathit{PreR1}$, $\mathit{Pre}(R1 \fatsemi R2)$, which formally compute the preconditions for $R1$, $(R1 \fatsemi R2)$ are given below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PreR1}} \\
\hline
\mathit{DecomposedMission} \\
\hline
\exists\, r? : \mathit{Resource} \bullet \\
\mathit{STATUS}(r?) = \mathit{Idle} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Pre}(R1 \fatsemi R2)} \\
\hline
\mathit{DecomposedMission} \\
\hline
\exists\, r? : \mathit{Resource} \bullet \\
\mathit{STATUS}(r?) = \mathit{Idle} \land \\
r? \notin \mathrm{dom}\, \mathit{ASSIGN} \\
\hline
\end{array}
$$

The left hand side of (i) evaluates to

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Mission} \land \mathit{PreR1}} \\
\hline
\mathit{DecomposedMission} \\
\hline
\mathrm{dom}(\mathit{STATUS} \rhd \mathit{Busy}) = \mathrm{dom}\, \mathit{ASSIGN} \land \\
\exists\, r? : \mathit{Resource} \bullet \\
\mathit{STATUS}(r?) = \mathit{Idle} \\
\hline
\end{array}
$$

To prove (a) we must show that the constraint $r? \notin \mathrm{dom}\, \mathit{ASSIGN}$ in the schema $\mathit{Pre}(R1 \fatsemi R2)$ can be derived from the constraints of the schema $\mathit{Mission} \land \mathit{PreR1}$. Below we show how this can be done: Since a resource can be either Busy or Idle, we can write $\mathit{STATUS}(r?) = \mathit{Idle} \Leftrightarrow \mathit{STATUS}(r?) \neq \mathit{Busy}$. When the integrity constraint $\mathrm{dom}(\mathit{STATUS} \rhd \mathit{Busy}) = \mathrm{dom}\, \mathit{ASSIGN}$ is satisfied, $\mathit{STATUS}(r?) \neq \mathit{Busy} \Leftrightarrow r? \notin \mathrm{dom}\, \mathit{ASSIGN}$.

Proof of (b):

The precondition schemas $\mathit{PreC1}$ and $\mathit{Pre}(C1 \fatsemi C2)$ which formally compute the preconditions of $C1$ and $C1 \fatsemi C2$ are given below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PreC1}} \\
\hline
\mathit{DecomposedMission} \\
r? : \mathit{Resource} \\
\hline
\mathit{STATUS}(r?) = \mathit{Busy} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Pre}(C1 \fatsemi C2)} \\
\hline
\mathit{DecomposedMission} \\
r? : \mathit{Resource} \\
\hline
\mathit{STATUS}(r?) = \mathit{Busy} \\
r? \in \mathrm{dom}\, \mathit{ASSIGN} \\
\hline
\end{array}
$$

When the constraints in $\mathit{Mission}$ and $\mathit{PreC1}$ are satisfied, $\mathit{STATUS}(r?) = \mathit{Busy} \Rightarrow r? \in \mathrm{dom}\, \mathit{ASSIGN}$, and (b) is proved.

Semantic Atomicity Property

Proof Obligation:

Every correct partial semantic history containing one or more incomplete *Respond* or *Cancel* transaction is a prefix of a correct complete semantic history.

Proof:

The proof that the mission database has the semantic atomicity property proceeds by induction. In the mission example the transactions *Respond* and *Cancel* are decomposed into multiple sections. We refer to a transaction that has completed one or more but not all the sections as an incomplete transaction. Suppose we have n incomplete *Respond* and *Cancel* transactions. We show how an incomplete *Respond* or *Cancel* transaction can be completed, thereby decreasing the number of incomplete transactions by at least one. By repeated applications of our argument, it is possible to reduce the number of incomplete *Respond* or *Cancel* transactions to zero, at which point the history is a complete correct multilevel semantic history.

Consider an incomplete *Respond* transaction. If the *Respond* transaction has completed step $R1$, the preconditions of the next step $R2$ may or may not be satisfied. Suppose the precondition of the step $R2$ is not satisfied; this occurs only when a step $C1$ executes before the step $R1$, and the step $C2$ has not yet executed, and both the *Respond* and the *Cancel* transactions are executing on the same resource. In such a case, the precondition of the step $C2$ will be satisfied which can be executed. Moreover the postcondition of the step $C2$ will establish the precondition of step $R2$ which can now execute. In this way the *Respond* transaction can be completed. Using similar arguments, it can be shown how a *Cancel* transaction may be completed.

Consistent Execution Property

Proof Obligation:

For any complete semantic history H, if the initial state satisfies the constraints listed in schema *Mission*, then the final state also satisfies the those constraints.

Proof:

The database is in a consistent state when all the integrity constraints are satisfied. In the example, the database will be in a consistent state when $\mathrm{dom}(STATUS \rhd Busy) = \mathrm{dom}\, ASSIGN$. In other words, the database will be in a consistent state when $\mathrm{dom}(R1ax \uplus R2ax) = \varnothing \wedge \mathrm{dom}(C1ax \uplus C2ax) = \varnothing$.

Let $\widetilde{R1ax}$, $\widetilde{R2ax}$, $\widetilde{C1ax}$, $\widetilde{C2ax}$ be the values of the variables in the initial state of the history. In the initial state when no transactions have been executed the database is in a consistent state and the following relation holds: $\mathrm{dom}((\widetilde{R1ax} \uplus \widetilde{R2ax}) = \varnothing \wedge (\widetilde{C1ax} \uplus \widetilde{C2ax})) = \varnothing$.

Let $\widehat{R1ax}$, $\widehat{R2ax}$, $\widehat{C1ax}$, $\widehat{C2ax}$ be the values of the variables at the end of the complete history. Note that the variable $R1ax$ is updated in section $R1$ and variable $R2ax$ is updated similarly in section $R2$. Hence, when all the *Reserve* transactions have completed execution, the variables $R1ax$ and $R2ax$ are changed in the same way. Therefore, at the end of the complete history, $\mathrm{dom}((\widehat{R1ax} \uplus \widehat{R2ax}) = \mathrm{dom}((\widetilde{R1ax} \uplus \widetilde{R2ax})$

Similarly when all *Cancel* transactions have completed execution, $\mathrm{dom}((\widehat{C1ax} \uplus \widehat{C2ax}) = \mathrm{dom}((\widetilde{C1ax} \uplus \widetilde{C2ax})$

Thus at the end of the complete history, $\mathrm{dom}((\widehat{R1ax} \uplus \widehat{R2ax}) = \varnothing \wedge (\widehat{C1ax} \uplus \widehat{C2ax})) = \varnothing$. Therefore when all transactions have completed execution, the database is in a consistent state.

Sensitive Transaction Isolation Property

Proof Obligation:

The *Report* transaction outputs consistent data.

Proof:

The sensitive transaction isolation property is achieved by construction.

Consider the *Report* transaction. We compute the subset of the original invariants that must hold as a precondition for *Report*. For this case all the state variables in *Mission* are involved in the computation of outputs of *Report*. Thus all the original invariants of *Mission* must be satisfied for *Report* to generate consistent output. In other words, the original invariant $\mathrm{dom}(STATUS \rhd \{Busy\}) = \mathrm{dom}(ASSIGN)$ must be included as an explicit precondition for $Report1$. Instead of including such a complex precondition for *Report*, we include the more simpler precondition $R1ax = R2ax \wedge C1ax = C2ax$. We do this because simple preconditions are evaluated faster than complex ones. Note that $(R1ax = R2ax \wedge C1ax = C2ax)$ ensures that the original invariant $\mathrm{dom}(STATUS \rhd \{Busy\}) = \mathrm{dom}\, ASSIGN$ will be satisfied.

For the mission database we have no transactions in which outputs are generated in multiple steps. Thus the problem of outputs from later steps not being consistent with outputs from earlier steps does not arise in this example.

Appendix B – Automated Verification of the Mission Decomposition

In this appendix we show how we can automatically verify the mission decomposition. We use the SMV model checker [McM92, CGL94] for the automated analysis. The model checker cannot directly verify the mission decomposition as the specification does not represent a finite state machine. To solve this problem, we developed a finite state abstraction which can be verified by the model checker. In the following paragraphs we describe briefly the SMV model checker, the finite state abstraction of the mission database, the specification of the properties, the input and the output produced by the model checker.

The SMV Model Checker

The input to the symbolic model checker is a SMV program. The program contains declarations of state variables, the initial states of the variables, the transition relations that changes the state of the variables and the specification of the properties to be verified. A variable can be of the following types: boolean, scalar and fixed arrays. The SMV *init* and *next* functions define the initial value and the next-state value for a state variable. The *next* functions are usually specified with the *case* expression. A *case* expression returns the first expression on the right hand side of the colon (:), such that the corresponding condition on the left hand side is true. The properties to be verified must be specified as Computational Tree Logic (CTL) formulae. A CTL formula is a boolean expression, an existential (E) path formula, a universal (A) path formula, or the application of standard boolean operators to CTL formulae. A path formula is the application of the temporal operators next (X), eventually (F), or globally (G), to a CTL formula.

The SMV produces as output the result of verifying the properties. For each property, it either displays the result that the property holds, or it produces a counterexample illustrating the violation of the property.

Finite State Abstraction of the Mission Database

In the following paragraphs we describe a very simple finite model of the mission database.

State Variables

The finite model has two resources and three possible threats. The ids of the two resources are 1 and 2 respectively. The variable status gives the status information of each resource; status is specified as a fixed array. status[x] gives the status of resource x – the status of resource x can take on two values: 0 (indicating that resource x is *Idle*) or 1 (indicating that resource x is *Busy*). rr, cr denote the respective resource input to the *Respond* and *Cancel* transaction. When the input is *Respond* (*Cancel*), rr (cr) equals 1 or 2; at other times it is equal to 0. The input must be passed as parameters from one step to another. rp (cp) denotes the parameter that must be passed from section R1 (C1) to R2 (C2). rp is specified as a fixed array; if x is an input parameter that must be passed from section R1 to R2, rp[x] equals 1, otherwise it is 0. cp is specified similarly. The three possible threats are denoted by a, b and c; the threat input to the *Respond* transaction, denoted by rt, can take on any of these three values. The variable assign which stores the assignment of resource to threat information is specified as a fixed array. assign[x] denotes the threat to which resource x is assigned. When resource x is not assigned to any threat, the value of assign[x] is n.

The auxiliary variables r1ax, r2ax, c1ax, c2ax are specified as fixed arrays. r1ax[x] gives the number of times section R1 has executed with resource x as its input. r1ax[x] can take on any values in the range 0 to 3. r2ax[x] denotes the the number of times section R2 has executed with resource x as its input. c1ax, c2ax are specified similarly.

We have one variable input which enumerates all possible types of sections in the mission database. To ensure that the range of the auxiliary variable does not limit the number of sections that can be executed in the finite state model, we introduce a dummy section known as AuxInit. AuxInit decrements the value of the auxiliary variables in the finite state model but does not alter the database objects. The other sections are responsible for changing the auxiliary variables and the database objects.

To verify the sensitive transaction isolation property we need a flag variable, ReportJustExecuted, that indicates whether ReportD has been executed.

State Initialization

The initial state of each variable is specified by the *init* function. In the initial state when no transaction has executed r1ax[x] is set to 0; similarly, all other auxiliary variables are assigned to 0. Since all resources are idle in the initial state and no resource is assigned to any threat, status[x] and assign[x] are assigned the values 0 and n respectively. Initially no ReportD has been executed, and so ReportJustExecuted is initialized to N.

State Transformation

The value of the variable is changed according to the *next* function. The next function is specified by a case expression. Each step which updates a state variable contribute at least one case statement in the expression. The left hand side of the colon (:) is an expression which is conjunction of the following: (i) the name of the step activating the state change, (ii) the preconditions of the step, (iii) if no direct assignment is made to the variable, extra preconditions to check that the variable lies within the specified range, (iv) a precondition to check that the counter variable is within the specified range. The right hand side of the colon (:) specifies how the variable must be updated if the left hand expression is satisfied. The last statement in the case expression specifies the default value of the variable in case none of the previous cases have been satisfied. The left hand side of the last case statement is 1; if the state variable represents a flag variable the right hand side equals N, otherwise the right hand side specifies the current value of the variable.

The initialization and state transformation of the variable status[1] is given below.

```
init(status[1]) := 0;
next(status[1]) := case
    (input = R1) & (rr = 1) & (status[1] = 0) & (r1ax[1] < maxaux) : 1;
    (input = C1) & (cr = 1) & (status[1] = 1) & (c1ax[1] < maxaux) : 0;
    1 : status[1];
    esac;
```

Specifying the Generalized Integrity Constraints and Properties in CTL

Generalized Integrity Constraints

The generalized integrity constraints must be satisfied by all states of the model and so they are specified as invariants. Below we describe how each generalized integrity constraint is mapped to the corresponding CTL formula.

(i) Each resource can be either busy or idle. The definition of status as a fixed array ensures this constraint; the variable status[x] denoting the status of rooms x allows status[x] to take only one value: 0 or 1.

(ii) Each resource can be assigned to at most one threat. The definition of assign as a fixed array takes care of this constraint; assign[x] can take only one value which means that at most one guest can be assigned to a room.

(iii) The set of resources with status busy and the set of resources whose assignment has been canceled equals the set of resources assigned to threats together with the set of resources whose status has been changed to busy but which are not yet assigned to threats.

```
SPEC AG((status[1] | (c1ax[1] - c2ax[1] > 0)) <->
     (!(assign[1] = n) | (r1ax[1] - r2ax[1] > 0)))
SPEC AG((status[2] | (c1ax[2] - c2ax[2] > 0)) <->
     (!(assign[2] = n) | (r1ax[2] - r2ax[2] > 0)))
```

Semantic Atomicity Property

The semantic atomicity property ensures that when a transaction has been partially executed, then it will eventually complete. Semantic atomicity involves showing that in any state, if the preconditions of R2 or C2 are not satisfied, then in that state there are no outstanding incomplete *Respond* or *Cancel* transactions. The corresponding CTL formula is given below.

```
SPEC AG(!((rp[1] & (assign[1] = n) & (r1ax[1] > r2ax[1]) & (r2ax[1] < maxaux))
     | (rp[2] & (assign[2] = n) & (r1ax[2] > r2ax[2]) & (r2ax[2] < maxaux))) &
     !((cp[1] & !(assign[1] = n) & (c1ax[1] > c2ax[1]) & (c2ax[1] < maxaux)) |
     (cp[2] & !(assign[2] = n) & (c1ax[2] > c2ax[2]) & (c2ax[2] < maxaux)))
     -> (r1ax[1] = r2ax[1]) & (r1ax[2] = r2ax[2]) &
     (c1ax[1] = c2ax[1]) & (c1ax[2] = c2ax[2]))
```

In the next section we show the input that was presented to the SMV model checker.

Consistent Execution Property

The consistent execution property requires the original integrity constraints (specified in Section 2.1) to hold after the completion of all transactions. Note that the first two invariants specified in Section 7 correspond to the original integrity constraints and they hold in every state in the decomposed specification. Only the last invariant (iii) corresponds to a generalized integrity constraint. Thus for the consistent execution property we need to show that when all the *Respond* and *Cancel* transactions complete, the original integrity constraints corresponding to only item (iii) is satisfied.

Before specifying the consistent execution property we need to show the specification of the original integrity constraints corresponding to item (iii). This is as follows:

(iii) The set of resources with status *Busy* equals the set of resources assigned to threats. The CTL formula is:

```
SPEC AG(status[1]  <-> !(assign[1] = n))
SPEC AG(status[2]  <-> !(assign[2] = n))
```

When all *Respond* transactions complete, the following conditions hold: (i) r1ax[1] = r2ax[1], (ii) r1ax[2] = r2ax[2], (iii) c1ax[1] = c2ax[1], and (iv) c1ax[2] = c2ax[2].

The specification of the consistent execution property in CTL is given below.

```
SPEC AG(((r1ax[1] = r2ax[1]) & (r1ax[2] = r2ax[2]) & (c1ax[1] = c2ax[1])
     & (c1ax[2] = c2ax[2])) ->
     ((status[1]  <-> !(assign[1] = n)) &
     (status[2]  <-> !(assign[2] = n))))
```

Sensitive Transaction Isolation Property

The sensitive transaction isolation property involves verifying that transaction *Report* is not executed when there are any incomplete *Respond* or *Cancel* transaction. The CTL formula shown below specifies this property.

```
SPEC AG((!(r1ax[1] = r2ax[1]) | !(r1ax[2] = r2ax[2]) |
     !(c1ax[1] = c2ax[1]) | !(c1ax[2] = c2ax[2]))
     -> (ReportJustExecuted = N))
```

Input to the SMV Model Checker

```
MODULE main

VAR
 input : {R1,R2,C1,C2,ReportD,AuxInit};
 status : array 1..2 of boolean;
 assign : array 1..2 of {a,b,c,n};
 r1ax :array 1..2 of 0..3;
 r2ax :array 1..2 of 0..3;
 c1ax :array 1..2 of 0..3;
 c2ax :array 1..2 of 0..3;
 rr : {0,1,2} ;
 cr : {0,1,2} ;
 rp : array 1..2 of boolean;
 cp : array 1..2 of boolean;
 rt : {a,b,c} ;
 ReportJustExecuted : {Y,N};

SPEC AG((status[1] | (c1ax[1] - c2ax[1] > 0)) <->
     (!(assign[1] = n) | (r1ax[1] - r2ax[1] > 0)))
SPEC AG((status[2] | (c1ax[2] - c2ax[2] > 0)) <->
     (!(assign[2] = n) | (r1ax[2] - r2ax[2] > 0)))

SPEC AG(((r1ax[1] = r2ax[1]) & (r1ax[2] = r2ax[2]) & (c1ax[1] = c2ax[1])
     & (c1ax[2] = c2ax[2])) ->
     ((status[1]  <-> !(assign[1] = n)) & (status[2]  <-> !(assign[2] = n))))

SPEC AG((!(r1ax[1] = r2ax[1]) | !(r1ax[2] = r2ax[2]) |
     !(c1ax[1] = c2ax[1]) | !(c1ax[2] = c2ax[2])) -> (ReportJustExecuted = N))

SPEC AG(!((rp[1] & (assign[1] = n) & (r1ax[1] > r2ax[1]) & (r2ax[1] < maxaux))
     | (rp[2] & (assign[2] = n) & (r1ax[2] > r2ax[2]) & (r2ax[2] < maxaux))) &
     !((cp[1] & !(assign[1] = n) & (c1ax[1] > c2ax[1]) & (c2ax[1] < maxaux))
     | (cp[2] & !(assign[2] = n) & (c1ax[2] > c2ax[2]) & (c2ax[2] < maxaux)))
     -> (r1ax[1] = r2ax[1]) & (r1ax[2] = r2ax[2]) & (c1ax[1] = c2ax[1]) &
     (c1ax[2] = c2ax[2]))

DEFINE
 maxaux := 3;

ASSIGN

 init(status[1]) := 0;
 next(status[1]) := case
   (input = R1) & (rr = 1) & (status[1] = 0) & (r1ax[1] < maxaux) : 1;
   (input = C1) & (cr = 1) & (status[1] = 1) & (c1ax[1] < maxaux) : 0;
   1 : status[1];
   esac;

 init(status[2]) := 0;
 next(status[2]) := case
   (input = R1) & (rr = 2) & (status[2] = 0) & (r1ax[2] < maxaux) : 1;
   (input = C1) & (cr = 2) & (status[2] = 1) & (c1ax[2] < maxaux) : 0;
   1 : status[2];
   esac;

 init(r1ax[1]) := 0;
 next(r1ax[1]) := case
   (input = R1) & (rr=1) & (status[1] = 0) & (r1ax[1] < maxaux) : r1ax[1] + 1;
   (input = AuxInit) & (r1ax[1] >= r2ax[1]) : r1ax[1] - r2ax[1];
   1: r1ax[1] ;
```

```
  esac;

init(r1ax[2]) := 0;
next(r1ax[2]) := case
  (input = R1) & (rr=2) & (status[2] = 0) & (r1ax[2] < maxaux) : r1ax[2] + 1;
  (input = AuxInit) & (r1ax[2] >= r2ax[2]) : r1ax[2] - r2ax[2];
  1: r1ax[2] ;
  esac;

init(c1ax[1]) := 0;
next(c1ax[1]) := case
  (input = C1) & (cr=1) & (status[1] = 1) & (c1ax[1] < maxaux) : c1ax[1] + 1;
  (input = AuxInit) & (c1ax[1] >= c2ax[1]) : c1ax[1] - c2ax[1];
  1 : c1ax[1];
  esac;

init(c1ax[2]) := 0;
next(c1ax[2]) := case
  (input = C1) & (cr=2) & (status[2] = 1) & (c1ax[2] < maxaux) : c1ax[2] + 1;
  (input = AuxInit) & (c1ax[2] >= c2ax[2]) : c1ax[2] - c2ax[2];
  1 : c1ax[2];
  esac;

init(r2ax[1]) := 0;
next(r2ax[1]) := case
  (input = R2) & rp[1] & (assign[1] = n) & (r1ax[1] > r2ax[1]) &
    (r2ax[1] < maxaux) : r2ax[1] + 1;
  (input = AuxInit) : 0;
  1: r2ax[1] ;
  esac;

init(r2ax[2]) := 0;
next(r2ax[2]) := case
  (input = R2) & rp[2] & (assign[2] = n) & (r1ax[2] > r2ax[2])
    & (r2ax[2] < maxaux) : r2ax[2] + 1;
  (input = AuxInit) : 0;
  1: r2ax[2];
  esac;

init(c2ax[1]) := 0;
next(c2ax[1]) := case
  (input = C2) & !(cp[1] = 0) & !(assign[1] = n) & (c1ax[1] > c2ax[1])
    & (c2ax[1] < maxaux) : c2ax[1] + 1;
  (input = AuxInit) : 0;
  1 : c2ax[1];
  esac;

init(c2ax[2]) := 0;
next(c2ax[2]) := case
```

```
    (input = C2) & !(cp[2] = 0) & !(assign[2] = n) & (c1ax[2] > c2ax[2])
      &  (c2ax[2] < maxaux) : c2ax[2] + 1;
    (input = AuxInit) : 0;
    1 : c2ax[2];
    esac;

  init(assign[1]) := n;
  next(assign[1]) := case
    (input = R2) & rp[1] & (assign[1] = n) & (r1ax[1] > r2ax[1]) &
      (r2ax[1] < maxaux) : rt;
    (input = C2) & cp[1] & !(assign[1] = n) & (c1ax[1] > c2ax[1])
      & (c2ax[1] < maxaux) : n;
    1 : assign[1];
    esac;

  init(assign[2]) := n;
  next(assign[2]) := case
    (input = R2) & rp[2] & (assign[2] = n) & (r1ax[2] > r2ax[2])
      & (r2ax[2] < maxaux) : rt;
    (input = C2) & cp[2] & !(assign[2] = n) & (c1ax[2] > c2ax[2])
      & (c2ax[2] < maxaux) : n;
    1 : assign[2];
    esac;

  init(ReportJustExecuted) := N;
  next(ReportJustExecuted) := case
    (input = ReportD) & (r1ax[1] = r2ax[1]) & (r1ax[2] = r2ax[2])
      & (c1ax[1] = c2ax[1]) & (c1ax[2] = c2ax[2]) : Y ;
    1 : N;
  esac;

  init(rr) := 0;
  next(rr) := case
    (input = R1) : {1,2};
    1 : 0;
  esac;

  init(cr) := 0;
  next(cr) := case
    (input = R1) : {1,2};
    1 : 0;
esac;

  init(rp[1]) := 0;
  next(rp[1]) := case
    (input = R1) & (rr=1) & (status[1] = 0) & (r1ax[1] < maxaux) : 1;
    (input = R2) & (rp[1]) & (assign[1] = n) & (r1ax[1] > r2ax[1]) &
      (r2ax[1] < maxaux) :  (!(r1ax[1] - r2ax[1] - 1 = 0));
    1 : rp[1];
  esac;

 init(rp[2]) := 0;
 next(rp[2]) := case
  (input = R1) & (rr=2) & (status[2] = 0) & (r1ax[2] < maxaux) : 1;
  (input = R2) & (rp[2]) & (assign[2] = n) & (r1ax[2] > r2ax[2]) &
   (r2ax[2] < maxaux) : (!(r1ax[2] - r2ax[2] - 1 = 0));
  1 : rp[2];
  esac;

 init(cp[1]) := 0;
 next(cp[1]) := case
  (input = C1) & (cr=1) & (status[1] = 1) & (c1ax[1] < maxaux) : 1;
  (input = C2) & (cp[1]) & !(assign[1] = n) & (c1ax[1] > c2ax[1]) &
   (c2ax[1] < maxaux) : (!(c1ax[1] - c2ax[1] - 1 = 0));
  1 : cp[1];
  esac;

 init(cp[2]) := 0;
 next(cp[2]) := case
  (input = C1) & (cr=2) & (status[2] = 1) & (c1ax[2] < maxaux) : 1;
  (input = C2) & (cp[2]) & !(assign[2] = n) & (c1ax[2] > c2ax[2]) &
   (c2ax[2] < maxaux) : (!(c1ax[2] - c2ax[2] - 1 = 0));
  1 : cp[2];
  esac;
```

The output produced by executing the SMV model checker on this input is presented next.

7.1 Output Produced by the SMV Model Checker

```
-- specification AG (status[1] | c1ax[1] - c2ax[1] > 0 <-... is true
-- specification AG (status[2] | c1ax[2] - c2ax[2] > 0 <-... is true
-- specification AG (r1ax[1] = r2ax[1] & r1ax[2] = r2ax[2... is true
-- specification AG (!r1ax[1] = r2ax[1] | !r1ax[2] = r2ax... is true
-- specification AG (!(rp[1] & assign[1] = n & r1ax[1] > ... is true

resources used:
user time: 30.1833 s, system time: 0.35 s
BDD nodes allocated: 54881
Bytes allocated: 1703936
BDD nodes representing transition relation: 5098 + 1
```