USING FORMAL METHODS IN SEMANTIC-BASED DECOMPOSITION OF TRANSACTIONS

by
Indrakshi Ray
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
the Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_______________ Paul Ammann, Dissertation Director

_______________ Sushil Jajodia, Dissertation Director

_______________ Richard Carver

_______________ Jeff Offutt

_______________ Carl Harris, Interim Dean for Graduate Studies and Research

_______________ Lloyd Griffiths, Dean, School of Information Technology and Engineering

Date: _______ Summer 1997
George Mason University
Fairfax, Virginia

Using Formal Methods in Semantic-Based Decomposition of Transactions

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University.

By

Indrakshi Ray
B.E. Computer Science and Technology, Calcutta University, India, June 1988
M.S. Computer Science and Engineering, Jadavpur University, India, March 1991

Directors: Dr. Paul Ammann, Associate Professor and Dr. Sushil Jajodia, Professor
Information and Software Systems Engineering

Summer 1997
George Mason University
Fairfax, Virginia

Dedication

I dedicate this thesis to my Gurudev, Sri Sri Mohonananda Brahmachari, whose blessings made this work possible.

Acknowledgments

I take this opportunity to thank my advisors Professor Paul Ammann and Professor Sushil Jajodia for their advice, support, and encouragement during my pursuit of the Ph.D. program. Professor Ammann's technical expertise and cheerful disposition together with Professor Jajodia's exceptional mature insight made my work easier. I consider myself fortunate for having such advisors and I look forward to the day when I will be able to provide such able guidance to my students.

I thank Professor Offutt and Professor Carver for their valuable suggestions and for serving as members of my doctoral advisory committee. I thank all those teachers whose encouragement motivated me to do well.

I thank my friends and colleagues here at George Mason, Vijayalakshmi Atluri, Claudio Bettini, Csilla Farkas, Jia-lin Ling, Padmaja Mavuluri, Liu Peng, Indrajit Ray, Mahua Roy, and Pierangela Samarati who encouraged me and helped me achieve this goal.

I thank my parents for all their love, support, and sacrifice; my sister for her encouragement and support. I thank my husband for his love and continuous support and for patiently bearing with me during this period. I also thank my in-laws, relatives and friends for all their support.

Table of Contents

List of Tables

List of Figures

Figure **Page**

Abstract

USING FORMAL METHODS IN SEMANTIC-BASED DECOMPOSITION OF TRANSACTIONS

Indrakshi Ray, Ph.D.

George Mason University, 1997

Dissertation Directors: Dr. Paul Ammann and Dr. Sushil Jajodia

The need for transaction decomposition arises in a variety of different application domains. For example, long duration transactions may be decomposed to improve performance, global transactions in multidatabases may be decomposed to preserve local database autonomy, and multilevel secure transactions may be decomposed to avoid leaking sensitive information. However transaction decomposition achieves these various objectives by sacrificing those desirable properties that form the foundation of syntactic correctness approaches such as conflict serializability. This makes the traditional transaction processing model which uses a syntactic based correctness criterion unsuitable for processing applications that require transactions to be decomposed. This thesis proposes a model of transaction decomposition based on using the semantic information of an application. The model is suitable for the application domains stated above.

Decomposition of a transaction into steps violates of the atomicity, consistency and isolation properties. Once these properties are violated we lose the traditional basis for assessing the correctness of executions of transactions. Our first contribution is the development of a set of replacement properties that are specifically designed

for reasoning about decompositions. These properties form the basis of semantic correctness and ensures that database consistency is preserved, transactions output consistent data, and all partially executed transactions complete. The exact details of the semantic correctness properties depend on the domain being addressed. In this thesis, we focus on three application domains, namely, the long duration transaction domain, the multidatabase domain, and the multilevel secure domain, and specify the semantic correctness properties appropriately for each of these domains.

The semantic correctness properties are defined in terms of serial execution of steps of transactions. To improve the performance, steps of transactions must be executed concurrently. Our second contribution, therefore, is proposing notions of correctness for concurrent executions of steps and developing appropriate mechanisms to achieve these. For each application domain we propose concurrency control mechanisms which produce correct concurrent executions and at the same time preserve appropriate domain specific requirements such as performance, autonomy, or security. In the long duration case, our model provides greater concurrency and allows a larger class of histories than the traditional model. In the multidatabase case, our model achieves greater autonomy, provides greater concurrency, and allows a larger class of histories. In the multilevel secure case, our approach allows a larger class of multilevel transactions to be processed, requires lesser synchronization between security levels, and achieves more concurrency.

Our third contribution lies in demonstrating how example applications can be analyzed to show the existence of the semantic correctness properties. Each of these properties generates a set of proof obligations that must be successfully discharged to ensure the correct execution of the application. Convincing formal arguments must be made to discharge these proof obligations; this motivated us to explore suitable specification languages for capturing an application's semantics and specifying the properties. The use of formal specification languages allow for standard formal analysis techniques of verification and refinement.

We investigated a number of specification languages. Among these we found Object Z to be the closest to fulfilling our needs, although it fell short of being adequate. We demonstrate how example applications from each application domain can be analyzed using Object Z.

Since the Object Z specification language does not have adequate tool-support, all the analysis using Object Z was performed manually. For large complex applications hand analysis may not be feasible. Our fourth contribution involves investigating the applicability of existing mechanized software verification tools for automated analysis of applications. The two common approaches of automated software verification are theorem proving and model checking. For each of these approaches we used example tools – the B-Toolkit (for the theorem proving approach) and the SMV model checker (for the model checking approach) – and discovered the relative merits and demerits of each of these approaches as they apply to the verification of semantic-based decompositions.

Chapter 1

Introduction

Database systems are one of the most important applications of computer technology. They provide an organized and uniform method for handling large volumes of data involved in applications such as banking systems and airlines reservation systems.

A database consists of a set of objects; the values of these objects comprise the state of the database. Integrity constraints or invariants are predicates that specify the relationships among the objects. A database state is said to be consistent if the values of objects satisfy the integrity constraints, otherwise the state is inconsistent. Users perform operations on databases using special application programs known as *transactions*.

The traditional model [BHG87] is built on three properties*, namely, *atomicity*, *consistency*, and *isolation*. Atomicity is the all or nothing property which means that all the operations of a transaction must be treated as a single unit – either all the operations are executed, or none are. Consistency means that a transaction must take the database from one consistent state to another. Isolation requires that no transaction should view the partial effects of any other concurrently executing transaction. Traditional transaction processing mechanisms implement the traditional model syntactically, namely, with conflict serializability. The traditional transaction processing

*A fourth property, durability, is necessary, but we do not discuss it as it is not of interest to this thesis.

model has proved to be a commercial success for many applications.

1.1 Problem Statement and Thesis

Problem Statement

The traditional transaction processing model is not adequate for certain applications having special requirements.

Thesis

1. A semantic-based model is more appropriate for at least three such application domains, namely, long duration transaction domain, multidatabase domain, and multilevel secure database domain.

2. Concurrency control mechanisms tailored to each specific domain can efficiently implement the semantic based model.

3. The analysis required in semantic based model is feasible for applications of interest.

In the following, we discuss why the traditional model fails in the above mentioned three application domains. In Section 1.3, we summarize our contributions.

1.2 Where the Syntactic Approach 'Fails'

The traditional transaction processing model is not suitable for certain application domains. For example, this model is not appropriate for applications with long duration transactions owing to performance reasons. The autonomy requirement in a multidatabase makes this model unsuitable for multidatabase applications. Finally, the security requirement in a multilevel secure database makes this model inappropriate for multilevel secure applications.

Consider the problems with long duration transactions. A long duration transaction [GMS87] takes a long time to execute, typically in the order of hours or days. The traditional model requires transactions to satisfy the isolation property. If isolation is to be ensured, the data items accessed by the long duration transaction are unavailable to other transactions for long periods of time; this in turn delays these other transactions. This delay not only lowers the throughput but also increases the probability of deadlocks. The probability of deadlocks is further increased owing to the large number of data items accessed by long duration transactions.[†] To resolve a deadlock one or more transactions must be aborted. Finally, the long execution time makes a long duration transaction more vulnerable to system crashes, thereby causing it to abort. When a long duration transaction aborts, all the work done by the transaction is wasted. This, in turn, further lowers the transaction throughput. Thus using the traditional transaction model for long duration transactions reduces performance to such extent as may be unacceptable.

Consider the problems with multidatabase applications. A multidatabase, which is an integrated collection of autonomous heterogeneous databases, poses the following four, often contradictory, requirements. First, local databases require both design autonomy to accommodate the diverse legacy nature of the local databases, and execution autonomy to ensure that local transactions are not unduly blocked by global transactions (transactions that run on more than one database). Second, management of global transactions must be distributed to avoid bottlenecks and tolerate failure in the global database. Third, integrity constraints must be maintained,

[†]The probability of deadlock is considered to be proportional to the fourth power of the number of locks acquired by the data item [GR93].

not only on each local database, but also on the global database. Finally, concurrent processing of transactions requires that execution histories be correct, an objective traditionally achieved with the syntactic criterion of serializability. The traditional transaction processing model does not meet these requirements. For example, consider the atomicity requirement of the traditional transaction processing model. The two-phase commit protocol, which is used to ensure the atomicity of global transactions, requires each local database to support the *prepared* state, which clearly violates design autonomy. Since local databases can no longer abort or commit global steps (parts of global transactions running on the local database) at their discretion, the two phase commit protocol also violates execution autonomy. The traditional transaction processing model, therefore, is not adequate for processing multidatabase applications.

Finally, consider the problems with multilevel secure (MLS) database applications. An MLS database is associated with a partially ordered set of security levels (the ordering relation is known as the dominance relation); the objects and the transactions are also assigned security levels. Each object is assigned only one security level and a multilevel transaction is associated with a range of security levels. Multilevel transactions are often decomposed into smaller units or steps (in the literature [AJR96, CM92, CJ93] these are referred to as sections) such that each step is associated with only one security level. An MLS database has an added security requirement: no information can be passed either directly or indirectly from a dominating level to a dominated level. This security requirement directly conflicts with the atomicity requirement of the traditional transaction processing model. The chief difficulty is that a dominating step of a transaction may be unable to complete due to violations of the integrity constraints, and a rollback of this dominating step can be exploited to indirectly pass information to the dominated step. Thus, the traditional transaction processing model cannot be used to process MLS database applications for security reasons.

From the above discussion, it is evident that enforcing the atomicity, consistency and isolation properties of the traditional transaction processing model results in the violation of domain specific requirements. Consequently, to meet the specific requirements, these properties must be sacrificed to some extent. Sacrificing the properties, in turn, introduces a new set of problems. For example, sacrificing the consistency property may result in the database becoming inconsistent. Giving up isolation will expose each transaction to the partial effects of other transactions, thereby causing transactions to view inconsistent states. Relaxation of the properties, therefore, cannot be arbitrary and needs to be done in a careful and controlled manner. Once these properties, which form the basis of syntactic-based correctness criterion have been sacrificed, the problem is how to give assurance of the continued correct behavior of the application. Our model provides a way to relax the syntactic correctness properties and yet provide assurance of correct behavior.

1.3 Summary of Contributions

In all the application domains stated in Section 1.2 decomposing transactions into atomic steps have been found to solve most of the problems discussed. Decomposition enhances performance in the long duration domain, provides autonomy in the multidatabase domain and ensures security in the multilevel secure domain. However, decomposing transactions into steps results in a loss of the atomicity, consistency and isolation properties. Our first contribution is proposing a set of replacement properties, namely *semantic atomicity*, *consistent execution*, and *sensitive transaction isolation*, to remedy this loss. These properties are defined in terms of *semantic histories* which are necessary to reason about correct and incorrect interleavings of transactions.

The semantic atomicity property ensures that all partially executed transactions can complete, or, in other words, either all or none of the steps of a transaction appear in a history. Consistent execution ensures that if a correct complete semantic history executes in a consistent state, the final state is also consistent. Sensitive transaction isolation ensures that no inconsistencies are displayed to the user. These properties are more relaxed than their counterparts in the traditional model, and form the basis of semantic correctness. In addition to the replacement properties we specify a fourth property, the *composition property*, which ensures that the decomposition correctly models the original transaction.

The semantic correctness properties ensure that applications perform in a desirable and expected way. To achieve semantic correctness, interleaving of steps of different transactions must be controlled efficiently. Towards this end, we develop the *successor set mechanism*. The successor set mechanism in the multidatabase domain is also responsible for the distributed management of global transactions and for ensuring local autonomy.

The semantic correctness properties are defined over semantic histories in which the steps of transactions execute serially. To improve the throughput, steps of transactions must execute concurrently. Our second contribution is proposing notions of correct concurrent execution and developing concurrency control mechanisms for each of the above application domain. These mechanisms not only produce correct concurrent executions but also satisfy the domain specific requirements. In the long duration case, our model provides greater concurrency and allows a larger class of histories than the traditional model. Most of the other semantic-based models [AAS93, FÖ89, GM83, GMS87, KS94] proposed in the long duration domain assume that the application developer is responsible for coming up with a correct decomposition and do not provide any guidance for performing this decomposition or reasoning about its correctness. The related work in Section 2.1 provides a detailed comparison of our model with other models proposed in this area. In the multidatabase case, our model achieves greater autonomy, provides greater concurrency and allows a larger class of histories than the traditional model. The other models proposed in this domain are not satisfactory because some require a centralized global transaction manager [BS88, MRB+92, MRKS92], some make assumptions about the nature of integrity constraints [DE89, GM91], and others have high execution time overhead [BS88, MRB+92, RKS95]. Again, see Section 2.2 for details. In the multilevel secure case, our approach allows a larger class of multilevel transactions to be processed, requires lesser synchronization between security levels and achieves more concurrency than the approaches based on the traditional transaction processing model.

Our third contribution is in demonstrating how example applications can be analyzed to show the existence of the semantic correctness properties. The semantic correctness properties generate a set of proof obligations that must be successfully discharged. Convincing formal arguments must be made to discharge these proof obligations; this motivated us to investigate the use of a suitable specification language for capturing the semantics of the applications and specifying the semantic correctness properties as the use of a specification language allows for standard formal analysis techniques for verification and refinement. Although not adequate due to lack of tool support, the Object Z [DD90] specification language was the closest to fulfilling our needs. Using Object Z we can specify transactions, steps, integrity constraints, relaxed integrity constraints, sensitive transaction isolation property, consistent execution property, and composition property. Using example applications we show how typical applications from each application domain can be analyzed for satisfaction of the properties in Object Z.

Since the Object Z specification language does not have adequate tool-support, all the analysis using Object Z was performed manually. However, hand analysis is not plausible for large complex applications. Thus our fourth contribution involves investigating the applicability of existing mechanized software verification tools for automated analysis of applications. The two common approaches for automated software verification are theorem proving and model checking. Specifically, we looked at the B-Toolkit [Sor94] for the theorem proving approach and the SMV [McM92] for the model checking approach. The B-Toolkit allows the specification and verification of only those properties which are invariants. Using the B-Toolkit we could only check whether the decomposed specification satisfies the relaxed integrity constraints. This verification, however, was not completely automated as it required aid from the user in the form of user theories. Moreover, since the B-Toolkit does not support temporal features, we could not specify or verify the semantic correctness properties. The SMV model checker, on the other hand, does not have this shortcoming; the properties verified by the SMV must be specified in Computational Tree Logic (CTL) which is a branching time temporal logic. However, model checkers can verify only finite state machines; software, in general, are infinite state machines and cannot be directly verified by model checkers. This required us to create finite state abstractions of the decomposed application which could be verified by the model checker. On the finite state abstraction of the example we studied we could verify all of the semantic correctness properties.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 discusses related works in the area of long duration transactions, multidatabase applications, and multilevel transactions. Chapter 3 presents our basic semantic-based transaction processing model. Chapters 4, 5 and 6 adapts this model to specific application areas – long-duration transactions, multidatabase applications, and multilevel secure applications respectively. These chapters also provide concurrency control mechanisms appropriate for the specific models. Chapter 7 describes how the analysis of applications can be automated using existing software verification tools. Chapter 8 summarizes the contribution and discusses future work. Examples of how long duration applications, multidatabase applications and multilevel applications can be decomposed and analyzed using the Object Z specification language appears in Appendix A, B and C respectively. The detailed specification of an example application that was verified by the B-Toolkit and the sample outputs produced by the B-Toolkit at different stages of the verification process are listed in Appendix D. Finally, Appendix E lists the input to the model checker (consisting of the model and the specification of the properties) and the outputs produced by the model checker (indicating the result of verification of the properties on the given model).

Chapter 2

Related Work

The related work has been divided into three sections on the basis of the application domain.

2.1 Related Work in Long Duration Transactions

Most transaction-oriented models enforce a very low level, syntactic notion of consistency, namely serializability with respect to read/write conflicts [BHG87]. Some researchers [Her87, HW91, LMWF94, Lyn83, Wei84, Wei88] have expanded on this notion by classifying the data objects into abstract data types; transactions access the data objects through operations on the data types. Commutativity of these operations, and not the read/write operations, is used to determine conflicts between transactions, resulting in more concurrency. Still further concurrency can be achieved if recoverability [BR92] of operations is used instead of commutativity. Although we also adopt the theory of abstract data types for the purpose of defining correctness of transaction execution, there is a significant difference in the two approaches. Unlike [Her87, HW91, Wei88], we define the entire database as a state, with the transactions being the operations on the state. Furthermore we are interested in achieving more concurrency by expanding the set of correct execution histories such that some transactions need not be atomic.

Other extensions to the traditional transaction model include generalizing the notion of serializability and proposing new correctness criteria for transaction executions [AAS93, FÖ89, GM83, KS88, KS94, LKS91a, LKS91b, WR92]. In some of these works, researchers have decomposed transactions into steps [AAS93, BL96, FÖ89, GM83, JM87, KS88, KS94, SLJ88, SSV92] and developed semantic based correctness criteria [AAS93, BL96, FÖ89, GM83]. Researchers have variously introduced the notions of transaction steps, countersteps, allowed versus prohibited interleavings of steps, and implementations in locking environments.

Shasha, Simon and Valduriez [SSV92] describe how a transaction can be decomposed into steps, without using any semantic knowledge of the transactions. The authors use the standard notion of serializability as the correctness criterion. The authors show that if the steps of the transactions in a history are serializable, then the history is equivalent to a serial history. Unlike our work, the decomposition of transactions into steps is based on conflict information only. Also, certain decompositions are valid in our model that are not allowed by their system. Specifically, the decomposition of the *Reserve* transaction into steps $R1$, $R2$ and $R3$ in the Hotel Database shown in Chapter 7 will not be a correct decomposition according to their model.

Sha et al. [SLJ88] propose a notion of correctness based on database partitioning and transaction decomposition. The database is partitioned into *atomic data sets* using the knowledge of consistency constraints. The transactions are decomposed into *elementary transactions* using only the semantic information associated with a transaction. An elementary transaction must maintain the consistency constraints of the accessed atomic data sets. The authors introduce a correctness criteria known as *generalized setwise serializable*, in which it is required that the effect of execution of the elementary transactions on any atomic data set is equivalent to a serial execution of the elementary transactions on the data set. Generalized setwise serializable schedules maintain database consistency and also satisfy the post conditions of the transactions.

Agrawal and others [AAS93] propose three correctness criteria: *consistency*, *orderability* and *strong orderability* for non-serializable executions. The data objects in their model are classified into various abstract data types. These objects are accessed through user programs that are characterized by the presence of *consistency assertions*. Consistency assertions are predicates on database objects that are required to hold before an operation accessing the data objects is executed. The correctness criterion, consistency, is based on the users specifications and allows nonserializable executions that are acceptable to the users. The other two correctness criteria, orderability and strong orderability, are generalizations of view serializability and conflict serializability respectively. The user is responsible for writing correct programs and for specifying the correct consistency assertions in the user programs.

The ConTract model [WR92] is a formal model for specifying and controlling long-lived complex applications like office automation, CAD applications, etc. Consistency is not based on serializability but on application dependent correctness criteria. A ConTract is composed of a number of steps that are elementary units of work; the control flow of the steps is specified by a script. A ConTract is allowed to externalize its partial results before it completes. Compensation of a ConTract takes place when a user wants to abort a ConTract that has committed some of its steps. Corresponding to each step in the ConTract, a compensating step is specified which must be executed when the user cancels a ConTract. Cascading compensations may occur in the ConTract model. The motivation for our model is different; our model is suitable for applications consisting of long duration transactions. Thus, we do not provide a generalized model for modeling complex activities requiring cooperation among different transactions.

Korth and Speegle [KS88, KS94] introduce a transaction model called NT/PV (nested transactions with predicates and versions) model for long duration transactions. The NT/PV model incorporates the idea of nested transactions, multiple versions and explicit predicates to increase concurrency. A transaction, denoted by (T, P, I, O), is characterized by the set of subtransactions T constituting the transaction, the partial order P among the subtransactions, the input conditions or preconditions I and the output conditions or postconditions O. An execution of a transaction is characterized by a pair (R, X), where R is a run time dependency relation between the subtransactions of the transaction, and X denotes the state associated with the transaction. An execution of a transaction is NT/PV correct if (i) the subtransactions are executed in a state that satisfies the preconditions of the subtransactions, and (ii) the post conditions of the last subtransaction satisfy the post conditions of the transaction. An execution of an interleaved set of transactions is NT/PV correct if every transaction in the set executes correctly. The authors show how classical transactions and various notions of serializability can be represented by their model and propose an implementation which generates only NT/PV correct executions.

Though a transaction in our model can be represented as a NT/PV transaction, our notion of correct execution of a set of transaction is very different from that proposed by Korth and Speegle [KS94]. In the NT/PV model, the post conditions of top level transactions imply the satisfaction of database integrity constraints. Also one of the requirements for correct executions is that the post conditions of the last subtransaction of a transaction should satisfy the transaction's post conditions. This implies that in a correct execution, the last subtransaction of a top level transaction should satisfy the database integrity constraints. In our notion of correct execution of a set of transactions (which we call *correct semantic history*) we do not require the post condition of the last step of a transaction to satisfy the database integrity constraints. Note, however, that we do have other requirements for a correct execution. For example, we require the post condition of any step of any transaction to satisfy the modified invariants (which are the relaxed integrity constraints) and that after the execution of all steps of all the transactions, the database be in a consistent state.

The mechanism proposed by Korth and Speegle for generating NT/PV correct executions requires support for multiple versions. As our transaction model does not deal with multiple versions, and our notion of correct execution is different from that proposed by the authors, our algorithm also differs.

Our work is closely related to that of Farrag and Özsu [FÖ89] and hence we discuss this work at a more detailed level. In this work the authors introduce the notion of *breakpoint sets* associated with the steps of transactions. The breakpoint set of a step S_{ij} of a transaction T_i gives the types of transactions that may interleave transaction T_i after step S_{ij}. The burden of specifying breakpoint sets lies with the application developer and the authors assume that the developer will define a correct breakpoint set description. A schedule is *correct* if it is stepwise serial and for every transaction T_i that appears between steps S_{jk} and $S_{j(k+1)}$ of another transaction T_j, T_i is in the breakpoint set of the step S_{jk}. To identify a larger class of allowable schedules, called *relatively consistent schedules*, the authors propose a *precedence graph*. For a schedule to be relatively consistent, the precedence graph of the schedule must satisfy two criteria: (i) the precedence graph must be acyclic, and (ii) there must be a topological sort of the graph that gives a correct schedule. As illustrated in Section 3.7.1, our model requires only the first constraint.

The authors describe a lock based mechanism that generates only relatively consistent schedules. The mechanism uses four types of locks – relatively shared, relatively exclusive, shared and exclusive. Before reading or writing a data item, the appropriate lock (shared or exclusive) is acquired. The lock on an item can be acquired only if the transaction is in the breakpoint set of the most recently executed steps of the transactions holding relatively shared or relatively exclusive lock on the data item. After a step completes, all shared locks and exclusive locks acquired by this step are converted to relatively shared and relatively exclusive locks respectively. After a transaction terminates, a release graph is checked to see whether the locks of the transaction can be released. In addition to its inherent complexity, the mechanism requires breakpoint sets to satisfy an additional assumption that may be difficult to obtain in practice.

Our successor set notion is similar to the breakpoint sets proposed by Farrag and Özsu [FÖ89]. However, the semantics of the successor set differs from that of breakpoint sets. The most significant difference is that successor sets are sets of types of *steps*, whereas breakpoint sets are sets of types of *transactions*. Successor sets have the flexibility to express that type of step T_{km} is in the successor set of type of step T_{ij} even though the type of some other step T_{kl} of T_k may not be in the successor set of type of step T_{ij}.

The mechanism we propose is simpler than that proposed by Farrag and Özsu [FÖ89]. We use only two kinds of locks - shared and exclusive as opposed to the four kinds of locks in the mechanism proposed by the authors. Before reading or writing a data item, the appropriate lock (shared or exclusive) has to be acquired. A lock can be acquired only if the type of the current step is in the successor set of all the active transactions that had locked the data item. The lock release process is also very different. The locks acquired by a step are released when the step commits. In contrast, the earliest time when locks acquired by a step can be released in the mechanism proposed by the authors [FÖ89] is after the entire transaction terminates. Our mechanism can be adapted more easily to a standard two-phase locking environment.

Garcia-Molina [GM83] classified transactions into a number of *semantic types*, associated *compatibility sets* with each of these types, and defined the notion of *sensitive transactions* and *semantically consistent schedules*. A transaction is sensitive if it has to output consistent database data to be viewed by the user. A semantically consistent schedule is one such that it maintains the consistency of the database, and any sensitive transaction in it obtains a consistent view of the data. In a semantically consistent schedule, the steps of transactions belonging to the same compatibility set can be interleaved arbitrarily, whereas steps of transactions that do not belong to the same compatibility set can be interleaved as much as permitted by the serializability constraints.

The idea of using compensation has also been explored [GM83]. If a transaction T cannot be successfully completed, compensating steps are executed to compensate for the updates caused by the transaction. This avoids the aborts of compatible transactions that used the data produced by the partial execution of T. In this model, corresponding to each step s_i of a transaction, except the last, the application developer specifies a counterstep c_i. The counterstep c_i semantically undoes the effect of the step s_i. The countersteps are initiated by the system when a transaction aborts. For reasons of performance the authors state that countersteps should not be delayed or aborted. The burden of classifying the transactions into types, decomposing the transaction into steps, providing the compatibility sets, and designing countersteps rests with the application developer.

The Saga model [GMS87] also addresses long duration transactions. A Saga consists of a collection of transactions, each of which maintains database consistency. However any partial execution of the Saga is undesirable; either all the transactions in a Saga complete successfully or compensating transactions should be run to amend for the partial execution of the Saga. Thus corresponding to every transaction in the

Saga, except the last one, a compensating transaction is specified. The compensating transaction semantically undoes the effect of the corresponding transaction. Note that the role of compensating transactions in the Saga is different than the role of compensating steps in our model. In a Saga, each transaction as well compensating transactions always maintain database consistency. In our model, on the other hand, execution of step violates database consistency; compensating steps are therefore required to restore consistency.

A formal treatment of compensating transactions is presented by Korth, Levy and Silberschatz [KLS90]; in this work the authors propose a generalized model which allows the definition of various types of compensating transactions. In subsequent works [LKS91a, LKS91b] the authors discuss the role of compensation in the context of distributed transactions. The authors [LKS91b] discuss how the notion of atomicity can be relaxed in a distributed environment. A distributed transaction can be decomposed into subtransactions or steps where each step executes at a site. The relaxed atomicity of distributed transaction requires either (i) all steps of the distributed transaction to commit or (ii) all committed steps to be compensated for and all uncommitted steps to be aborted. Based on this notion of relaxed atomicity, the authors [LKS91a] present an optimistic 2PC protocol.

The idea of decomposing transactions into steps using semantic information and formally verifying the decomposition has been proposed [BL96]. The authors describe how a transaction can be formally specified by annotating the transaction with assertions [Hoa69]; these assertions are predicates that capture the conditions that must be true before and after the execution of the transaction. When transactions are decomposed into steps, the interstep assertions specify the conditions that are true between the execution of consecutive steps. These assertions are always satisfied when a transaction executes without interleaving. However, interleaving of transactions may result in the violation of interstep assertions. To prevent invalidation of interstep assertions, the interleaving of transactions must be controlled. Towards this end, the authors do a design time analysis to detect all potential interleavings that may cause violation of interstep assertions. The number of such analysis to be done depends on the number of possible interleavings; in other words, it is exponential in the number of the transactions and the number of steps each transaction is decomposed.

A schedule that is conflict equivalent to one in which the steps execute serially and respects the allowable interleavings is a semantically correct schedule. The authors propose an assertional concurrency control mechanism for generating semantically correct schedules. In addition to the conventional read and write locks, this mechanism has assertional locks. The assertional locks blocks other steps which may violate the interstep assertions thereby ensuring that only steps that do not violate the relevant interstep assertions execute concurrently. Unlike the read and write locks which are held for the duration of the step, the assertional locks are held until the next step of the transaction begins execution.

Although the authors address the same problem of ensuring correct decompositions, the solutions proposed are different. Our analysis is done at the specification level, as opposed to the analysis proposed by the authors [BL96] which is done at the code level. Code level analysis is more appropriate for improving the performance of legacy systems, where as our analysis may be appropriate for systems which are being developed from scratch. Also the implementation we propose is different; we require only read and write locks. Our decomposition allows a larger class of schedules than that of [BL96]. It would be interesting to compare the performance of the two approaches on some real world long duration transactions.

ACTA [CR94] is a framework for formally specifying extended transaction models. ACTA provides components, (e.g. operations, significant events, conditions on events, transaction dependencies) for describing the transaction models. However, our model cannot be represented within the ACTA framework because ACTA conditions cannot be used to express the preconditions and postconditions of a semantic history. Thus extensions to the ACTA framework are required to represent our model.

2.2 Related Work in Multidatabase Applications

Several schemes [BS88, MRB+92] have been proposed to ensure global serializability in a multidatabase environment. Unfortunately, these algorithms require the existence of a centralized global transaction manager and have high run time overhead because each requires a site graph to be maintained.

Mehrotra and other researchers [MRKS92] present the *GTM commit protocol* for ensuring semantic atomicity [GM83]. Semantic atomicity requires that either all subtransactions of a global transaction commit or all subtransactions are undone (by aborting the subtransactions or by executing compensating subtransactions). The subtransactions of a global transaction are classified into compensatable, pivot, and retriable. Compensatable subtransactions can be compensated by executing compensating subtransactions, retriable subtransactions can be successfully executed if retried a sufficient number of times, and pivot subtransactions can be neither compensated nor retried. Although the authors [MRKS92] do not formally specify how to classify subtransactions, a separate work [NZ96] suggests that this classification can be done on the basis of the structure of the code and compensating code. For example, if the subtransaction code does not contain preconditions involving database variables, then it is retriable, otherwise not. Similarly if the compensating code does not contain preconditions involving database variables then the subtransaction is compensatable. The GTM commit protocol requires the subtransactions of a global transaction to be committed in the following order: compensatable, pivot and retriable. The GTM commit protocol gives the local DBMS the freedom to abort a subtransaction. If the local DBMS aborts a compensatable or a pivot subtransaction, the GTM is informed which then sends an abort message to all the sites executing the global transaction. On receiving this message the other sites either execute compensating subtransactions or abort the subtransactions depending on whether or not the subtransactions have committed. If the local DBMS aborts a retriable subtransaction, then this subtransaction is retried until it executes successfully. After completing a subtransaction successfully, the local DBMS must wait for a commit message from the GTM before the subtransaction can be committed.

There are several differences between our approach and the one proposed by Mehrotra et al. [MRKS92]. The major difference is that we do not require a centralized global transaction manager (GTM) which is responsible for coordinating the global transactions. We do not classify the subtransactions* or enforce any commit ordering for ensuring semantic atomicity; consequently, our semantic atomicity property is more general (See Theorem 1 in [RAJ]). In our model, the local databases have complete autonomy to abort or commit the subtransactions executing at the site. This is in contrast to the model proposed by Mehrotra [MRKS92] where the local databases have the freedom to abort a subtransaction, but must wait for a message from the GTM before committing the subtransaction. Finally, in Mehrotra's work [MRKS92] compensating subtransactions are not considered part of the original transaction; instead, they are subtransactions of a separate compensating transaction. This fails to ensure that inconsistencies are not displayed to the user. This motivates the authors to propose a new correctness criterion, called *SRC*, which

*In the subsequent chapters of this thesis we use the generic term 'step' to mean a decomposed unit of a transaction and do not use application domain specific terms like 'subtransaction' or 'section'.

is stronger than serializability, and a new concurrency protocol for ensuring SRC. The protocol requires that the GTM maintain a graph known as the transaction-site graph. To prevent non-serializable executions, only certain types of cycles are allowed in the transaction-site graph. This requires the edge insertion and deletion process to perform several checks. Thus maintaining this graph imposes additional run time overhead both in terms of space and time. Our approach incurs no significant run time overhead, although our approach does require that an application must be analyzed statically off-line to prove the properties.

Several researchers [DE89, GM91, RKS95, MRKS91] maintain that traditional notion of serializability is needlessly restrictive in a multidatabase environment and have proposed alternative correctness criteria. Du and Elmagarmid [DE89] proposed the notion of *quasi-serializability*. A quasi-serial history is one in which all the local histories are conflict serializable and for any two global transactions G_i and G_j there is a linear ordering such that if G_i precedes G_j in this ordering then all of G_i's operations precede any of G_j's operation in any local history containing operations of G_i and G_j. A history is quasi-serializable if it is equivalent to a quasi-serial history. Moreover, Du and Elmagarmid illustrate how the altruistic locking algorithm can be used to generate quasi-serializable histories. The problem with quasi-serializability is that it can handle only a limited class of global integrity constraints (the constraints associated with data replication) and requires that the subtransactions of the global transactions have no value dependencies. In contrast, we are able to handle applications with global integrity constraints and also applications having value dependencies between subtransactions of a global transaction.

Mehrotra et al. [MRKS91] introduce the notion of *two-level serializability* as the correctness criterion for multidatabase transactions. A schedule is two-level serializable (2LSR) if all the local schedules are serializable and the projection of global operations on each site is also serializable. To characterize the set of acceptable 2LSR schedules, the authors introduce the notion of strong correctness. A strongly correct schedule is one which preserves database consistency and all transactions view consistent data. To ensure that 2LSR schedules are strongly correct, the authors place restrictions on the nature of integrity constraints, the types of transactions to be executed, and the type of database objects accessed. For example to ensure strong correctness, global transactions should not have value dependencies and there should not be any constraint involving local and global objects. (A data object is classified as global or local depending on whether or not it is associated with a global integrity constraint.) Morpain et al. [MCFP96] argue that global transactions not having value dependencies is a restrictive criterion for ensuring strong correctness and propose the use of a flow-graph for handling such transactions.

Rastogi, Korth, and Silberschatz [RKS95] also argue that serializability is a stringent requirement for multidatabase applications and propose weaker correctness notions based on exploiting transaction semantics. The authors propose the use of regular expressions for specifying non-allowable interleavings. Any execution which does not have non-allowable interleavings is correct. It is assumed, however, that non-allowable interleavings are given by the user. We, on the other hand, show how users can analyze an application and find undesirable interleavings. The concurrency control algorithms proposed by Rastogi et al. [RKS95] require a centralized global transaction manager for coordinating global transactions. These algorithms are based on graphs and have excessive run-time overhead.

Garcia-Molina [GM91] advocates the use of the Saga model for handling multidatabase applications. The Saga model [GMS87] decomposes transactions into atomic units such that the integrity constraints are satisfied after the execution of each atomic unit. The Saga model is appropriate for applications without global integrity constraints. For applications with global integrity constraints, the Saga model can be used provided the global constraints can be generalized by the local constraints. However, this may not always be possible. Specifically, the example shown in Appendix B cannot be handled by the Saga model.

2.3 Related Work in Multilevel Transactions

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. Also this algorithm [CM92] requires the operations in a transaction to be ordered according to the security lattice structure. This requirement is obviated in the algorithm by Costich and Jajodia [CJ93]; the algorithm 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 in [CM92, CJ93] suitable for only a limited class of multilevel transactions – transactions satisfying isolation atomicity property[AJR96].

†The data items and transactions in a multilevel secure database are associated with security levels; low and high are two such security levels.

Blaustein and others [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 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 if a section commits then all other sections must also commit; complete atomicity corresponds to traditional atomicity. 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 and other researchers [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.

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.

Chapter 3

Our Model

In our model, a *database* is specified as a collection of objects, along with some *invariants* or *integrity constraints* on these objects. 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. The invariants are predicates defined over the objects. A database state is said to be *consistent* if the values of the objects satisfy the given invariants.

A *transaction* is an operation that transforms one database state to another. Associated with each transaction is a set of *preconditions* and a set of *postconditions*. A precondition limits the database states to which a transaction can be applied. A postcondition constrains the possible database states after a transaction completes. Postconditions also constrain outputs. Together, preconditions and postconditions must ensure that if a transaction executes on a consistent state, the result is again a consistent state.

3.1 Decomposing a Transaction

Instead of executing a transaction as an atomic unit, we wish to break a transaction into steps and execute each of these steps as an atomic unit.

Definition 3.1.1 [Transaction Decomposition] A transaction T_i can be decomposed into a partially ordered set* of steps $\{S_{i1}, S_{i1}, \ldots, S_{in}\}$ with the ordering relation $\ll_i$ specified on the steps. For any two steps S_{ij} and S_{ik} of transaction T_i such that $S_{ij} \ll_i S_{ik}$, then all operations of S_{ij} must precede any operation of S_{ik}.

In place of the transaction, the steps execute atomically and in an order consistent with $\ll_i$. A transaction that has been decomposed into two or more steps is referred to as a *multistep* transaction.

One possible approach to decomposition is to treat the steps as transactions. In particular, one could insist that the integrity constraints hold after each step, which is the decision taken in the Saga model [GMS87]. However such a requirement is too strong for some applications, and so we develop a more flexible approach.

Relaxing the requirement that each step maintain database consistency introduces new problems. The execution of a step no longer preserves the integrity constraints and steps of other transactions are exposed to inconsistent states. This is problematic in some cases. For example, if transactions which output data execute in a state where the integrity constraints do not hold, inconsistencies may be displayed to the user. Once the integrity constraints are violated, the formal basis for assessing the correctness of subsequent behavior collapses. To solve this problem, we need a formal model that can accommodate the notion that some – but not all – violations of the integrity constraints are acceptable. The next section describes this model.

---

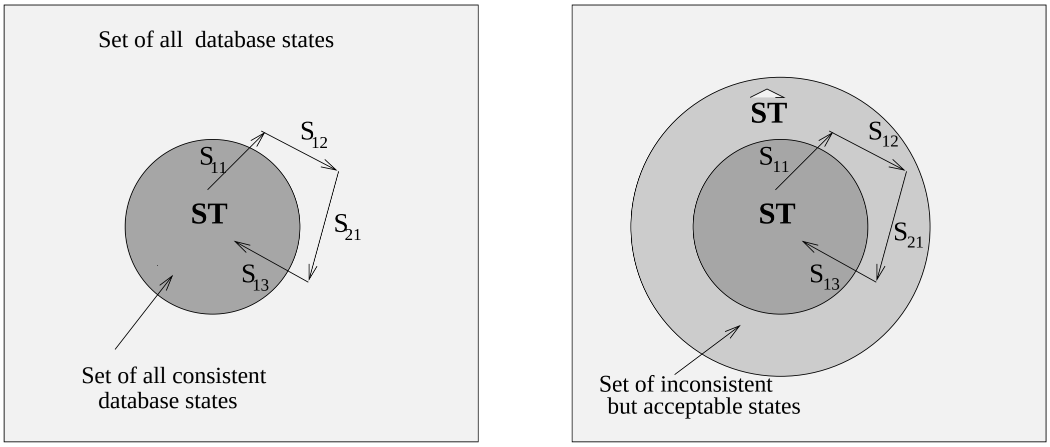
*Note that this is more general than that proposed by Ammann and others [AJR97] where a transaction is decomposed into a sequence of steps.

3.2 Generalizing the Original Integrity Constraints

A decomposition may yield steps that leave the database in a state in which the integrity constraints do not hold. The arrow labeled S_{11} in figure 3.1(a) illustrates this possibility. Figure 3.1(b) illustrates a model that allows inconsistent states – as defined by the integrity constraints – that are nonetheless acceptable. The temporary inconsistency introduced by S_{11} is allowed, and steps of some other transactions, for example S_{21}, may tolerate the inconsistency introduced by S_{11} and so are allowed to proceed. The general approach is to relax the original set of integrity constraints and decompose transactions such that each step satisfies the set of relaxed or generalized integrity constraints. The model in figure 3.1(b) has many advantages, including greater concurrency among steps. We formalize the model as follows.

Let I denote the original integrity constraints, and let $\mathbf{ST}$ denote the set consisting of all consistent states; that is, $\mathbf{ST} = \{ST : ST \text{ satisfies } I\}$. In the standard model, a transaction T_i always accesses a consistent state ST, where $ST \in \mathbf{ST}$. If ST_i denotes the state after the execution of T_i, then ST_i is also in $\mathbf{ST}$. When T_i is broken up into steps $S_{i1}, \ldots, S_{in}$, each step S_{ij} executes atomically. If ST_{ij} represents the state resulting from the partial execution of T_i through step S_{ij}, it is possible that ST_{ij} no longer satisfies the integrity constraints I and so lies outside $\mathbf{ST}$. Figure 3.1(a) illustrates this possibility.

In our approach, we define a set of generalized integrity constraints $\hat{I}$ by relaxing the original integrity constraints I. We decompose each transaction such that execution of any step results in a state that satisfies $\hat{I}$. Let $\widehat{\mathbf{ST}} = \{$ ST : ST satisfies $\hat{I}\}$. The relationship between $\mathbf{ST}$ and $\widehat{\mathbf{ST}}$ is shown in figure 3.1(b). The inner circle denotes $\mathbf{ST}$ and the outer circle denotes $\widehat{\mathbf{ST}}$ (signifying that $\mathbf{ST} \subset \widehat{\mathbf{ST}}$). The ring denotes the set of all states that satisfy $\hat{I}$ but not I. The important part about figure 3.1(b) is that the set of inconsistent but acceptable states is formally identified and



(a) Standard classification of database states

(b) Database states as classified in our model

Figure 3.1: Classification of the Database States

distinguished from the states that are unacceptable. The advantage is that formal analysis can be used to investigate activities in $\widehat{\mathbf{ST}}$.

The task of generalizing the integrity constraints is application specific and will be illustrated later while presenting the examples in the subsequent chapters. We use auxiliary variables to generalize the integrity constraints. Auxiliary variables are a standard method of reasoning about concurrent executions [OG76] and, in particular, have been applied to the problem of semantic-based concurrency control [GM83, Appendix C].

With the notion of generalized integrity constraints in place, we now describe compensation in the context of our model.

3.3 Compensating Steps

A transaction may not complete successfully if a precondition of a step is not satisfied, if the user aborts the transaction, or if the system crashes. Incomplete transactions pose special problems in semantic oriented models because steps may commit before it is determined whether the transaction can complete.

Suppose a transaction T_1 aborts after its first step S_{11} commits. Some mechanism must undo the effects of S_{11}. Nullifying the effects of S_{11} using the backwards recovery method of traditional *undo* [BHG87], where the state that existed before S_{11} is physically restored, is not possible because steps of other transactions may have read the updates of S_{11}. Instead we adopt the forward recovery solution of a compensating step [GM83, GMS87], which semantically undoes the effects of the committed step S_{11} and does not disturb transactions that may have read from S_{11}.

Like other steps of a transaction, compensating steps execute atomically. However, the role of a compensating step differs from that of other steps. A compensating step is not considered part of the normal processing of a transaction; it is initiated when a partially executed transaction is aborted to semantically undo a transaction.

Note that some steps are not compensatable and some steps do not require any compensation. Moreover a compensating step is never required for the last step of a transaction; this is because when the last step commits, the transaction completes and it can no longer be aborted. Corresponding to each type of step which is compensatable, except those which occur last in a transaction, the application developer must specify a type of compensating step. A compensating step C_{ij} semantically undoes the effects of step S_{ij}.† With the notion of compensation in place, we now

†Note that the approach proposed by Ammann and others [AJR97] where a compensating step C_{ij} semantically undoes the cumulative effects of step $S_{i1}, S_{i2}, \ldots, S_{i(j-1)}$ is problematic for distributed applications; in a distributed application a compensating step executes at one site only and cannot

specify a transaction as a partially ordered set of steps and compensating steps which are ordered by the relation $\ll_i$. Since no operations of a compensating step C_{ij} will execute before any operation of a step S_{ij}, it is always true that $S_{ij} \ll_i C_{ij}$.

3.4 Semantic Histories

To reason about correct and incorrect interleavings we need the notion of semantic histories. But before giving this notion we need some definitions.

First we distinguish between the *type* of a step and an *instance* of a step. An application has a finite number of types of steps and compensating steps. Histories, defined subsequently, reflect actual transactions, and must reference instances of steps and compensating steps. A history may contain many instances of a step of a given type. In cases where it not necessary to distinguish the role of steps from that of compensating steps, we use the term 'step' generically and denote an instance of either a step or a compensating step of transaction T_i as T_{ij}. Where the roles differ, we use S_{ij} to denote an instance of a step and C_{ij} to denote an instance of a compensating step. The type of an instance of a step T_{ij} is denoted by $ty(T_{ij})$.

Definition 3.4.1 [Stepwise Serial History] A *stepwise serial history* H over a set of transactions $\mathbf{T} = \{T_1, \ldots, T_m\}$ is a sequence of steps and compensating steps ordered by the relation $\ll_H$ such that

1. a step T_{ij} either appears exactly once in H or does not appear at all,
2. for any two steps S_{ij} and T_{ik} of some $T_i \in \mathbf{T}$, $S_{ij} \ll_H T_{ik}$ if $T_{ik} \not\ll_i S_{ij}$,
3. if step $T_{ik} \in H$, then $S_{ij} \in H$, for all steps S_{ij} such that $S_{ij} \ll_i T_{ik}$,

undo the effect of steps executed at other sites.

4. if a compensating step $C_{ij} \in H$, then $S_{ij} \in H$ and $S_{ij} \ll_H C_{ij}$,

5. if $S_{ij} \ll_H S_{im}$ and $C_{ij} \in H$, then $C_{im} \in H$ and $C_{im} \ll_H C_{ij}$.

Condition (1) ensures that every step or compensating step of a transaction occurs at most once. Condition (2) ensures that the order of the steps in a transaction is preserved. Condition (3) ensures that for every T_{ik} in a stepwise serial history all steps of T_i preceding T_{ik} must also be present in the history. Condition (4) ensures that a compensating step appears after the corresponding step. Condition (5) ensures that if the steps of a transaction are executed in a particular order, the corresponding compensating steps must be executed in the reverse order.

Definition 3.4.2 [Complete Execution] Consider a transaction T_i decomposed into steps $S_{i1}, \ldots, S_{in}$ with compensating steps C_{i1}, …, C_{in}. The execution of T_i in a history H is a *complete* execution if either (i) all n steps of T_i appear in H or (ii) some steps of T_i, namely, S_{i1}, …, S_{ij} appear in H followed by the corresponding compensating steps $C_{ij}, \ldots, C_{i1}$ where $j < n$.

The sequences $S_{i1}, \ldots, S_{in}$ and $S_{i1}, \ldots, S_{ij}, C_{ij}, \ldots, C_{i1}$ are examples of *execution sequences* [GM83] of transaction T_i. The sequence $S_{i1}, \ldots, S_{in}$ is a *successful* execution sequence of T_i, and the sequence $S_{i1}, \ldots, S_{ij}, C_{ij}, \ldots C_{i1}$ is an *unsuccessful* execution sequence of T_i.

A stepwise serial history, as given by Definition 3.4.1, lists the steps of transactions comprising the history; it does not capture the state of the database before and after the execution of each step in the history. This state information is needed to reason about correct and incorrect executions. For this reason we introduce the notion of *semantic histories*.

Definition 3.4.3 **[Semantic History]** A *semantic* history H is a stepwise serial history bound to

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

We use the term partial semantic history informally for cases in which the execution of at least one transaction actually is incomplete, but from a formal perspective, partial semantic histories are just semantic histories. Complete semantic histories are a special case of a semantic histories.

Definition 3.4.4 **[Complete Semantic History]** A semantic history H over a set of transactions $\mathbf{T}$ is a *complete* semantic history if the execution of each T_i in $\mathbf{T}$ is *complete*.

Next we define what it means for a semantic history to be correct.

Definition 3.4.5 **[Correct Semantic History]** A semantic history H is a *correct semantic history* if

1. the initial state is in $\mathbf{ST}$,
2. the states before and after each step in H are in $\widehat{\mathbf{ST}}$, and
3. the precondition for each step is satisfied in the corresponding state.

Definition 3.4.6 **[Correct Complete Semantic History]** A complete semantic history H is a *correct complete semantic history* if

1. H is a correct semantic history, and

2. the final state is in **ST**.

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

3.5 Properties of a Decomposition

To ensure the correct behavior of an application in which transactions have been decomposed into steps, we formulate a set of necessary properties, namely, composition property, consistent execution property, sensitive transaction isolation property, and semantic atomicity property. We consider a decomposition to be valid if it satisfies the necessary properties. Along with the necessary property we also describe an optional property, namely, the successful execution property, which may be required for certain applications.

3.5.1 Composition Property

The first necessary property, the *composition property*, relates steps in a decomposition to the original transaction.

Composition Property Let $S_{i1}, \ldots, S_{in}$ be the steps of transaction T_i and ST be a state that satisfies the original integrity constraints I. Then executing the steps in an order consistent with $\ll_i$ and in isolation on ST produces the same final state as executing T_i on ST.

3.5.2 Sensitive Transaction Isolation Property

In our model, we allow transactions to access database states that do not satisfy the original integrity constraints (that is, states in $\widehat{\mathbf{ST}} - \mathbf{ST}$). But we may wish to keep some transactions from viewing any inconsistency with respect to the original integrity constraints. For example, some transactions may output data to users; these transactions are called *sensitive* transactions [GM83]. We require sensitive transactions to appear to have generated outputs from a consistent state.

Sensitive Transaction Isolation Property All output data produced by a sensitive transaction T_i should have the appearance that it is based on a consistent state in $\mathbf{ST}$, even though the steps of T_i may access database states in $\widehat{\mathbf{ST}} - \mathbf{ST}$.

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 step or steps that generate the outputs. Second, as pointed out by Rastogi, Korth, and Silberschatz [RKS95], if outputs are generated by multiple steps, interleavings between these steps must be controlled to ensure that outputs from later steps are consistent with outputs from earlier steps.

3.5.3 Consistent Execution Property

Similar to the consistency property for traditional databases, we place the following requirement on semantic histories:

Consistent Execution Property: If we execute a correct complete semantic history H on an initial state (i.e., the state prior to the execution of any step in H) that satisfies the original integrity constraints I, then the final state (i.e., the state after the execution of the last step in H) also satisfies the original integrity constraints I.

3.5.4 Successful Execution Property

When transactions have been broken up into steps, it may not be possible to complete a transaction successfully. This may happen if a precondition of some later step of the transaction is not satisfied. Also, the interleaving of steps of different transactions may result in a state from which it is not possible to successfully complete some transaction. The successful execution property ensures that such a situation is avoided; if a transaction has been partially executed, then it can complete without resorting to compensation.

Successful Execution Property Every correct semantic history H_p defined over a set of transactions $\mathbf{T}$ is a prefix of some correct complete semantic history H over $\mathbf{T}$ such that for each $T_i \in \mathbf{T}$ that is incomplete in H_p, H contains a successful execution sequence of T_i.

Unlike the other properties we have stated so far, successful execution is an optional property. Successful execution property requires that all the preconditions of a transaction should appear in the first step. A precondition check is often associated with the update of some object and it is necessary that the check and the corresponding update be performed atomically. Thus insisting on successful execution property may force too many operations in the first step of the transaction – which is undesirable from the performance point of view. Hence we do not insist that applications have the successful execution property.

3.5.5 Semantic Atomicity Property

When a transaction has been partially executed, that is, committed one or more steps, it may not be possible to complete the transaction. This may happen if the precondition of some later step is not satisfied, and it is not possible to semantically undo the effects of the committed steps by executing compensating steps. The *semantic atomicity property* given below ensures that such a situation is avoided. If a transaction has completed some, but not all, of its steps, it should be possible to complete the transaction. Unlike the successful execution property, which is optional, the semantic atomicity property is required.

Semantic Atomicity Property Every correct semantic history H_p defined over a set of transactions $\mathbf{T}$ is a prefix of some correct complete semantic history H over $\mathbf{T}$.

3.6 Successor Sets

After presenting the properties we now describe mechanisms to efficiently implement our model. For the purpose of generalizing the integrity constraints, the decomposition process introduces additional objects known as auxiliary variables. These additional objects are present primarily to support analysis. For efficient implementation, we want to avoid instantiating the objects. Enforcing the properties also impose constraints on the order of execution of steps; checking the satisfaction of such constraints is expensive and our goal is to avoid such checks in the implementation. Successor sets are the mechanism we use to achieve these objectives. Intuitively, a successor set mechanism is used to specify allowable interleavings and it obviates the need to implement auxiliary variables.

Definition 3.6.1 [Successor Set] The successor set of $ty(T_{ij})$, denoted $SS(ty(T_{ij}))$, is a set of types of steps.

At this point, the notion of successor sets is purely syntactic. Subsequently, we define the constraints under which a successor set description is correct with respect to a particular decomposition. But before giving the correctness characterization, we wish to define a correct successor set history. To achieve this goal we introduce the notion of conflict into our model. Two operations *conflict* if both operate on the same data item and at least one is a Write. Two steps T_{ij} and T_{pq} *conflict* if they contain conflicting operations. The definition of conflict allows us to define a notion of correctness with respect to successor set descriptions that is not overly restrictive.

Definition 3.6.2 [Correct Successor Set History] H is a correct successor set history if it satisfies the following conditions:

1. H is a correct semantic history.

2. If T_i is incomplete in the prefix of H that ends at T_{pq}, and T_{ij} is the last step in T_i such that

   (a) T_{ij} conflicts with T_{pq} and

   (b) T_{ij} precedes T_{pq} in H

   then $ty(T_{pq}) \in SS(ty(T_{ij}))$.

With respect to the specifications given with generalized integrity constraints, not all successor set descriptions are valid. Informally, a successor set is valid with respect to a specification containing generalized integrity constraints if any complete[‡]

[‡]A complete correct successor set history is one in which all the transactions have completed execution; a partial correct successor set history is one in which there is at least one incomplete transaction.

correct successor set history H_s is equivalent to a correct complete semantic history H_g generated by the specification containing the generalized integrity constraints. (Two histories are equivalent if they produce the same final state when executed on the same initial state.) Formally, we describe valid successor set descriptions with the *valid successor set property*:

[**Valid Successor Set Property**] A specification S_2 that employs a successor set description is valid with respect to specification S_1 with generalized integrity constraints if

1. S_2 satisfies the semantic atomicity property.

2. Any complete correct successor set history generated by S_2 is equivalent to a correct complete semantic history generated by S_1.

If an application requires the successful execution property, the successful execution property must be explicitly reverified on histories generated by S_2. The definition of valid successor property given above is more relaxed than that proposed by Ammann et al. [AJR97] to allow a larger class of correct successor set history.

Before concluding this section we introduce a helpful function that will be used later in Section 3.7.1 when presenting a graph-theoretic correctness characterization.

3.6.1 The Next Function

Consider the following scenario. Suppose that in a partial correct successor set history step T_{ij} of transaction T_i has been executed, and suppose we wish to execute step T_{pq}, where T_{pq} conflicts with T_{ij} or some step previous to T_{ij}. The *next* function, defined below, gives the earliest step in T_i after T_{ij} where T_{pq} is allowed to execute. The role of the next function in our model is similar to that of the F function proposed by Farrag and Özsu [FÖ89].

Definition 3.6.3 [**Next Function**] The *next function*, denoted by $NF(T_{ij}, T_{pq})$, gives the first step T_{ik} after T_{ij} in an execution sequence of T_i such that $ty(T_{pq}) \in SS(ty(T_{ik}))$. Formally, $T_{ik} = NF(T_{ij}, T_{pq})$ if the following conditions hold:

1. $ty(T_{pq}) \in SS(ty(T_{ik}))$ and

2. for each step T_{ir} such that $T_{ij} \ll_H T_{ir} \ll_H T_{ik}$, $ty(T_{pq}) \notin SS(ty(T_{ir}))$.

A transaction T_i may have multiple execution sequences, one successful, and others unsuccessful. The value returned by the next function $NF(S_{ij}, T_{pq})$ depends on the particular sequence of T_i. For example, for the successful execution sequence, $S_{i1}, \ldots, S_{ij}, \ldots, S_{ik}, \ldots, S_{in}$, $NF(S_{ij}, T_{pq}) = S_{ik}$. For the unsuccessful execution sequence, $S_{i1}, \ldots, S_{ij}, \ldots, S_{im}, C_{im}, \ldots, C_{i1}$, where $m < k$, $NF(S_{ij}, T_{pq}) = C_{il}$ $(l \in 1, 2, 3, \ldots, j)$. In our subsequent analysis, we consider histories where the execution sequence for each transaction is fixed, and so this nondeterministic nature of the next function is not an issue in our presentation.

For the purposes of our model, we constrain the next function with respect to conflict. Suppose step T_{pq} conflicts with T_{ij} or some earlier step of T_i and $ty(T_{pq}) \notin SS(ty(T_{ij}))$. Let $NF(T_{ij}, T_{pq}) = T_{ik}$. T_{pq} is not allowed to execute after steps T_{ij}, $T_{i(j+1)}$, ..., $T_{i(k-1)}$. But it is allowed to execute after step T_{ik}. This scenario is shown below.

$$T_{i1} \ \ldots \ T_{i(j-1)}\, T_{ij} \ \underbrace{T_{i(j+1)} \ T_{i(j+2)} \ \ldots \ T_{i(k-1)}}_{T_{pq} \ cannot \ interleavee} \, T_{ik} \ \overbrace{T_{i(k+1)} \ \ldots \ T_{in}}^{T_{pq} \ can \ interleave}$$

T_{ij} introduces some inconsistency that prevents T_{pq} from being executed. This inconsistency is resolved in step T_{ik}. If T_{ik} alters the database state in such a way that T_{pq} can execute, then T_{pq} and T_{ik} must conflict. We capture this property as an explicit constraint in our model. The specifier of a decomposition is obligated to verify the constraint against a successor set description.

Constraint 1 If T_{pq} conflicts with T_{ij} or some step in T_i previous to T_{ij} and $ty(T_{pq}) \notin SS(ty(T_{ij}))$, then T_{pq} also conflicts with $NF(T_{ij}, T_{pq})$.

Note that Constraint 1 ties the next function to conflict between steps. Put differently, Constraint 1 proscribes arbitrary successor set descriptions; there must be a reason for excluding a step from the successor set of another step.

3.7 Correct Stepwise Serializable Histories

For every pair of steps in a correct successor set history, all operations of one step appear before any operations of the other step. However if the steps of a transaction execute atomically and without any interleaving, the database system uses resources poorly. In this section we describe the notion of a *correct stepwise serializable history.* In a correct stepwise serializable history the steps of transactions need not be executed serially, but nevertheless the effect is the same as that of a correct successor set history. But first we define history in a manner similar to that proposed by Bernstein [BHG87].

Definition 3.7.1 [History] A history H defined over a set of transactions $\mathbf{T} = \{T_1, \ldots, T_m\}$, where each transaction T_i is composed of n_i steps, is a partial order of operations with ordering relation $\prec_H$ where:

1. $H = \cup_{i=1}^{m} \cup_{j=1}^{n_i} T_{ij}$;

2. $\prec_H \supseteq \cup_{i=1}^{m} \cup_{j=1}^{n_i} \prec_{ij}$; and

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

Condition (1) says that the execution represented by H involves precisely the operations of the steps of $\mathbf{T}$. Condition (2) says that H preserves the order of operations in each step ($\prec_{ij}$ is the relation that orders the operations in step S_{ij}). Condition (3) says that every pair of conflicting operations are ordered in H.

Two histories H and H' are said to be *conflict equivalent* if they are defined over the same set of steps, they have the same operations, and they order conflicting operations of steps in the same way. That is, for any pair of conflicting operations p_{ij} and q_{kl} belonging to T_{ij} and T_{kl} (respectively), if $p_{ij} \prec_H q_{kl}$, then $p_{ij} \prec_{H'} q_{kl}$.

A *correct stepwise serializable history* is one which is conflict equivalent to a correct successor set history. Next we build a graph model to reason about stepwise serializability of histories.

3.7.1 Acyclic Graph Model

In this section we show how to decide whether a history is a correct stepwise serializable history by analyzing a graph, called a precedence graph, derived from that history.

Definition 3.7.2 [Precedence Graph] Let H be a history defined over a set of transactions $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$. The *precedence graph* of H, denoted by $PG(H)$, is a directed graph where the nodes are the steps of the transactions in $\mathbf{T}$ and the edges are of three types constructed as follows:

1. Internal Edges – For each pair of ordered steps S_{ij} and T_{ik} of transaction T_i, such that $S_{ij} \ll_i T_{ik}$, there is an *internal* or I edge (S_{ij}, T_{ik}). (Note that S_{ij} is a step of T_i; T_{ik} is a step or compensating step of T_i.)

2. Conflict Edges – For each pair of conflicting steps T_{ij} and T_{pq} belonging to different transactions T_i and T_p, there is a *conflict* or C edge (T_{ij}, T_{pq}) if there is an operation in T_{ij} that conflicts with and precedes another operation in T_{pq}.

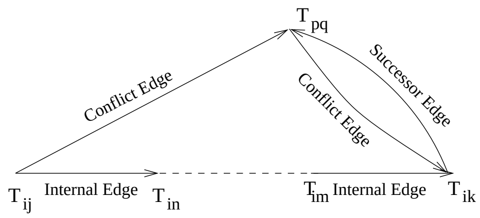


Figure 3.2: Edges of Precedence Graph involving T_{ij}, T_{ik}, T_{pq}

3. Successor Edges – For each pair of steps T_{ij} and T_{pq}, belonging to different transactions T_i and T_p respectively, such that there is a conflict edge from T_{ij} to T_{pq}, there is a *successor* or S edge $(NF(T_{ij}, T_{pq}), T_{pq})$.

Observation 1 *For a correct successor set history H, if there is an edge (T_{ij}, T_{pq}) in $PG(H)$ then T_{ij} precedes T_{pq}. It follows that for a correct successor set history H, the precedence graph $PG(H)$ is acyclic.*

Theorem 3.7.1 A history H is a correct stepwise serializable history iff $PG(H)$ is acyclic.

Proof: $\Leftarrow$: Since $PG(H)$ is acyclic it can be topologically sorted. Let H_c be any topological sort of $PG(H)$. We prove H_c is a correct successor set history by contradiction. Suppose H_c is not a correct successor set history. There must be some step T_{pq} which interleaves with transaction T_i such that T_{ij} is the last step in T_i conflicting with and preceding T_{pq}, and $ty(T_{pq}) \notin SS(ty(T_{ij}))$. Let $NF(T_{ij}, T_{pq}) = T_{ik}$. Since $ty(T_{pq}) \notin SS(ty(T_{ij}))$, T_{ik} is some step after T_{ij} and in $PG(H)$ there must be a successor edge (T_{ik}, T_{pq}) corresponding to the conflict edge (T_{ij}, T_{pq}). By Constraint 1 T_{pq} and T_{ik} conflict. As T_{ij} is the last operation which conflicts and precedes T_{pq}, T_{pq} must precede T_{ik}, and the conflict edge (T_{pq}, T_{ik}) is in H. As figure 3.2 shows, the result is a cycle in $PG(H)$ – a contradiction. The assumption that H_c is not a correct successor set history is wrong. Since H is equivalent to H_c, H is a correct stepwise serializable history.

$\Rightarrow$: Since H is a correct stepwise serializable history, it must be equivalent to some correct successor set history H'. H and H' have identical precedence graphs: $PG(H) = PG(H')$. From Observation 1, $PG(H')$ is acyclic. Hence $PG(H)$ is acyclic. $\square$

Chapter 4

Our Model for the Long Duration Domain

In this chapter we focus on the model for long duration transactions that execute on a centralized database; Appendix A in page 118 illustrates one such example application. (For distributed databases, the model proposed in the next chapter is more appropriate.) Database, transactions, integrity constraints, consistent and inconsistent states are defined as in Chapter 3.

4.1 Decomposition of Long Duration Transactions

For performance reasons we decompose a long duration transaction into a set of atomic steps. The decomposition is done such that each step performs a logical part of the work of a transaction.

4.2 Generalizing the Integrity Constraints

The decomposition of long duration transactions into steps violates the original integrity constraints. To provide a formal basis for arguing about the correctness of decompositions, we generalize the integrity constraints such that the execution of each step satisfies the generalized integrity constraints. This generalization is done using auxiliary variables. Auxiliary variables are distinct from the database objects; they are introduced to aid in the formal analysis.

4.3 Compensating Steps

As stated in Chapter 3, compensating steps are required to semantically undo the effects of partially executed transactions. For each multistep transaction we specify compensating steps for each type of step that is compensatable, except for those that occur last in a transaction.

4.4 Semantic Histories

Stepwise Serial Histories and Semantic Histories are defined as in Chapter 3.

4.5 Properties of the Decomposition

The composition property, consistent execution property, sensitive transaction isolation property, and semantic atomicity property must be verified for any application to be executed by our model. Successful execution is an optional property. Appendix A illustrates the Hotel Database, an example application in the long duration domain, and shows how the properties can be specified and verified using Object Z.

4.6 Concurrency Control Mechanism

In this section, we propose a concurrency control mechanism for our model.

We make the following assumptions:

1. Lock management is centralized.
2. The steps of a transaction are submitted in the correct sequence. That is an operation in step T_{rt} is submitted only after step T_{rs} commits where $T_{rs} \ll_r T_{rt}$.
3. If a transaction reads and writes the same data entity x, the read operation

precedes the write operation.

4. A transaction reads or writes an entity x at most once.

5. The algorithms specified below execute atomically.

Our mechanism uses two phase locking on the steps of the transactions. There are two modes in which a data item may be locked by a step - shared mode or exclusive mode. A step acquires an appropriate lock as a prerequisite for accessing a data item. A step is denied a lock if either another step holds a conflicting lock or if the step fails a test based on successor sets. Locks acquired by a step are released when the step commits or aborts.

For the purposes of this section, we define a step as a sequence of read and write operations followed by a commit or an abort operation,

$$T_{ij} =< O_{ij}(x_1), O_{ij}(x_2), \ldots, O_{ij}(x_n), E_{ij} >$$

where $O_{ij}(x)$ is either $R_{ij}(x)$ or $W_{ij}(x)$ and E_{ij} is either C_{ij} or A_{ij}, and a transaction is a sequence of steps followed by a termination operation,

$$T_i =< T_{i1}, \ldots, T_{in}, TR(T_i) >$$

We require the following data structures in addition to those required by the two phase locking protocol.

1. Active-Set – Set of Active Transactions

   Active-Set(x) – The active set for x keeps the list of all active transactions whose committed steps have accessed x. Whenever any step T_{ij} that reads or writes x commits, the transaction T_i is added to Active-set(x). After the transaction T_i terminates, T_i is removed from Active-Set(x).

2. Int-Set – Interleaving Sets

   Int-Set(T_i,x) – The interleaving set for x is associated with each active transaction T_i that accesses x. The interleaving set gives the types of the steps that can access the data item. If data item x has been accessed by step T_{ij} of T_i and T_{ij} or any step occurring after T_{ij} commits, then Int-set(T_i,x) is replaced by the successor set of the corresponding committed step.

4.6.1 Algorithms

Before a read operation can proceed, step T_{ij} needs a shared lock for x. There are two conditions for T_{ij} to acquire the shared lock: (i) No other step has an exclusive lock on x and (ii) T_{ij} is in Int-Set(T_k,x) for all active transactions T_k whose committed steps have accessed x. If either condition is not satisfied, the lock is not granted and step T_{ij} must try again later. When $R_{ij}(x)$ is retried, it must be re-executed from the first step of the algorithm.

Algorithm 4.6.1 *Algorithm for Read*

Procedure *Process-read ($R_{ij}(x)$)*

begin

    if *a step T_{lm} is holding an exclusive lock on x*

      exit;   */* Lock unavailable - T_{ij} can retry later */*

    for *each $T_k \in$ Active-set(x)*

        if *ty(T_{ij}) $\notin$ Int-set(T_k,x)*

          exit;   */* Lock unavailable - T_{ij} can retry later */*

    lock x in shared mode;

    accept($R_{ij}(x)$);

end

Before a write operation can proceed, step T_{ij} needs an exclusive lock for x. There are two conditions for T_{ij} to acquire the exclusive lock: (i) No other step has any lock on x and (ii) T_{ij} is in Int-Set(T_k,x) for all active transactions T_k whose committed steps have accessed x. If either condition is not satisfied, the lock is not granted and step T_{ij} must try again later. When $W_{ij}(x)$ is retried, it must be re-executed from the first step of the algorithm.

Algorithm 4.6.2 *Algorithm for Write*

Procedure *Process-write* $(W_{ij}(x))$

begin

    if *a step* T_{lm} *is holding any lock on* x

      exit;   */* Lock unavailable -* T_{ij} *can retry later */*

    for *each* $T_k \in$ *Active-set(x)*

        if $ty(T_{ij}) \notin$ *Int-set*(T_k,x)

            exit;   */* Lock unavailable -* T_{ij} *can retry later */*

    lock x *in exclusive mode*

    accept$(W_{ij}(x))$;

end

A step commits when all the operations of a step are complete. For each data item x locked by the transaction in the current or previous steps, the interleaving set associated with this transaction and data item x is replaced by the successor set of the step. The transaction is included in the list of active transactions that have accessed x. All locks acquired by this step are released.

Algorithm 4.6.3 <u>*Algorithm for Step Commit*</u>

Procedure *Process-stepcommit(C_{ij})*

begin

  for *each x locked by the transaction in this or previous step* **do**

    Int-set(T_i,x) $= SS(ty(T_{ij}))$ *;*

  for *each entity x locked by the transaction in this step* **do**

    begin

      if $T_i \notin$ *Active-set(x)*

        Active-set(x) $=$ *Active-set(x)* $\cup \{T_i\}$ *;*

      Release the lock on x which was acquired by T_{ij} ;

    end

end

A step may not always complete successfully; it may have to abort. The abort causes all the locks held by the step to be released. The abort of step T_{ij} does not affect the data structures Active-set(x) or Int-set(T_i,x); these data structures are adjusted with the transaction termination processing. Traditional recovery for aborted transactions, such as undo, is required for the aborted step, but details are omitted.

Algorithm 4.6.4 <u>*Algorithm for Step Abort*</u>

Procedure *Process-stepabort(A_{ij})*

begin

  / Restore values written by T_{ij} */*

  for *each entity x locked by the transaction in this step* **do**

Release the lock on x which was acquired by T_{ij}*;*

end

Termination removes a transaction from the set of active transactions. Since interleaving sets are associated only with active transactions, the interleaving set Int(T_i,x) is deleted when the transaction terminates.

Algorithm 4.6.5 *Algorithm for Transaction Terminate*

Procedure *Process-terminate(TR(*T_i*))*

begin

for *each entity x which was accessed by* T_i **do**

begin

Active-set(x) = Active-set(x) $-$ $\{T_i\}$;

delete the structure Int-set(T_i*,x) ;*

end

end

4.6.2 Correctness of the Mechanism

Theorem 4.6.1 Any history generated by our mechanism is a correct stepwise serializable history.

Proof: To prove that the history H generated by our mechanism is a correct stepwise serializable history we show that the precedence graph $PG(H)$ is acyclic and invoke Theorem 3.7.1. Constraint 1 is addressed by our mechanism as discussed above.

The assumptions of atomic lock release and a centralized lock manager ensure that lock release operations of steps form a total order, which we denote $\prec_{lr}$. Our mechanism submits step T_{rt} only after step T_{rs} commits when $T_{rs} \ll_r T_{rt}$. Thus all I edges in $PG(H)$ are consistent with $\prec_{lr}$. Since two phase locking controls concurrency between steps, all C edges in $PG(H)$ are also consistent with $\prec_{lr}$.

It remains to show that all S edges in $PG(H)$ are consistent with $\prec_{lr}$. Let $NF(T_{ij}, T_{pq}) = T_{ik}$ and consider the S edge (T_{ik}, T_{pq}). The reason for this S edge is that there is a C edge (T_{ij}, T_{pq}) and $ty(T_{pq}) \notin SS(ty(T_{ij}))$. The existence of the C edge provides two relevant facts: First T_{pq} and T_{ij} conflict on some x. Second, T_{ij} performs a Step Commit operation that releases the lock on x before T_{pq} can acquire the conflicting lock. Note that in the Step Commit operation, Int-set(T_i,x) is set to the $SS(ty(T_{ij}))$. Any request by T_{pq} for a conflicting lock on x through the Process-read or Process-write operations is rejected until step after T_{ik} commits and updates Int-set(T_i,x). Thus all S edges in $PG(H)$ are consistent with $\prec_{lr}$.

Since all I, C, and S edges in $PG(H)$ are consistent with the total order $\prec_{lr}$, $PG(H)$ is acyclic.

$\square$

Chapter 5

Our Model for the Multidatabase Domain

This chapter focuses on how our model must be adapted for multidatabase applications; Appendix B (page 131) illustrates an example multidatabase application and shows how verification obligations corresponding to this application can be discharged using Object Z.

A multidatabase consists of a collection of local databases.* At any given time, the local state is determined by the values of the objects in the local database. The global state is comprised of the states of all the individual local databases. A change in the value of a database object changes the state. Integrity constraints in a multidatabase application may be local or global; local constraints are defined over objects belonging to one local database and global constraints are defined over objects belonging to multiple local databases. The local state is consistent if the values of the objects satisfy the local integrity constraints. The global state is said to be consistent if both the local and global integrity constraints are satisfied. The objects in a multidatabase are accessed through transactions. A transaction is classified as local or global depending on whether the transaction accesses objects from one or more than one database.

*Unlike a distributed database, the local databases making up a multidatabase are heterogeneous and autonomous.

5.1 Decomposition of Global Transactions

To satisfy the autonomy requirements for multidatabases, it is necessary to execute a global transaction as a collection of local steps rather than as an atomic unit. A global transaction is decomposed into a partially ordered set of steps $\{S_{i1}, S_{i3}, \ldots, S_{in}\}$ where S_{ij} denotes the step of T_i that executes at site j. We assume that at most one step from a given global transaction executes at any site. A local transaction executes at one site only and can be thought to be composed of a single step; S_{kl} represents the step corresponding to the local transaction T_k executing at site l.

5.2 Generalizing the Global Integrity Constraints

When global transactions are decomposed into atomic steps, the global integrity constraints may not hold after the execution of a step. Once the global integrity constraints have been violated we have no formal basis for assessing the correctness of executions. The solution we adopt is to generalize the global integrity constraints such that the execution of each step satisfies the generalized integrity constraints. This generalization is done with the help of auxiliary variables. An auxiliary variable, like a database object, is associated with one site only.

5.3 Compensating Steps

When a global transaction aborts after committing one or more steps, the effect of the partially executed transaction is neutralized using compensating steps. For each type of compensatable step, except those that occur last in a transaction, the application developer must specify a compensating step. Since a compensating step semantically undoes the actions of a step, we assume that a step and the corresponding compensating step execute at the same site.

5.4 Semantic Histories

Stepwise serial history, semantic history, and correct semantic history are defined as in Section 3.4.3 of Chapter 3.

5.5 Properties of the Decomposition

All the properties, namely, composition property, sensitive transaction isolation property, consistent execution property, and semantic atomicity property need to be verified for multidatabase applications. The Banking Enterprise Database, an example application in the multidatabase domain, is formally specified using Object Z in Appendix B; the appendix also shows how the properties of the decomposition can be proved for this application.

5.6 Successor Sets

As in the long duration model, successor set descriptions obviate the need for implementing auxiliary variables or performing expensive run-time checks for ensuring the correct order of execution of steps. Moreover, in the multidatabase model, the successor sets help achieve local autonomy and distributed management of global transactions.

The successor sets in a multidatabase environment are distributed in nature. Each site in a multidatabase has an associated successor set which controls the interleavings of the steps of global transactions executing at that site. The successor set mechanism, therefore, is responsible for the distributed management of global transactions.

The successor set mechanism is also responsible for achieving local autonomy. In the absence of successor sets the interleaving is controlled by inserting additional preconditions in the steps of global transactions; if these preconditions involve objects at other sites, then the execution of a step is no longer a local operation and support for some atomic commit protocol is required for the atomic execution of steps. Thus without successor sets, we would not get local autonomy.

5.7 Concurrency Control Mechanism

For the purpose of coordinating the actions of the global transactions we need global transaction managers. However, unlike other approaches [MRKS92, RKS95], we do not require a centralized global transaction manager. Each site supports a global transaction manager (GTM) for coordinating the global transactions and steps submitted at that site. A global transaction manager in our model performs two functions. First, it produces stepwise serializable histories. Second, it coordinates the global transactions submitted at its site and achieves semantic atomicity. A global transaction manager assumes two roles: coordinator and participant. A global transaction manager acts as the coordinator for transactions submitted at its site; for transactions executing at this site, but which are coordinated by the GTM at some other site, it acts as a participant.

In the following paragraphs we discuss how the global transaction managers carry out these functions.

We make the following assumptions:

1. Each local database supports some concurrency control mechanism that produces serializable executions.

2. The coordinator of a transaction receives information about the ordering of the steps (that is, the partial order $\ll_i$), and the sites in which the steps must be executed.

3. Each database object is stored in one site only.

4. The algorithms specified below execute atomically.

We require the following data structures in addition to those required by the local concurrency control mechanism.

1. Active-Set – Set of Active Transactions

   Active-Set(j) – The active set of j keeps the list of all active transactions at site j. Whenever any step T_{ij} is submitted to the local scheduler at site j for execution, the transaction T_i is added to Active-Set(j). After the transaction T_i terminates, T_i is removed from the Active-Set(j).

5.7.1 Algorithms

Algorithms Executed by the Coordinator

The first task of the coordinator after receiving the transactions is distributing the steps of the transaction to the different participants. A step S_{ij} is sent to a participant only after all steps S_{im} preceding step S_{ij} in the partial order $\ll_i$ have completed execution.

Algorithm 5.7.1 *Algorithm for sending a step to a participant*

Procedure *Send-step* (S_{ij})
begin
    for *each uncommitted step* S_{im} *in* T_i **do**
        if $S_{im} \ll_i S_{ij}$
            exit *; /* retry sending step* S_{ij} *later */*
    send step S_{ij} *to site* j

end

Once a step has been sent to the appropriate participant, the next step is to wait for the result of the execution of that step. The coordinator either receives a failure message or a success message as the outcome. If the coordinator receives a failure message for a compensating step C_{ij}, it resends the compensating step to site j. If it receives a failure message for a step S_{ij} and T_i can always complete successfully, then the coordinator resends step S_{ij} to site j. On the other hand, if T_i cannot always complete successfully, the coordinator sends an abort message and compensating steps to all other participants that are executing or have already executed some step of T_i. When the coordinator receives a success message, it checks whether the transaction T_i has completed, in which case it sends a completion message to all participants involved in the execution of T_i.

Algorithm 5.7.2 *Algorithm for receiving results from participant*

Procedure *Receive-result (result(T_{ij}))*
begin
    if *result*$(T_{ij}) =$ *failure*
        if T_{ij} *is a compensating step or* T_i *can always complete successfully*
            resend step T_{ij} *to site j*
        else
            for *each site k that executed or is executing* S_{ik} **do**
                send abort message and compensating step C_{ik} *to site k*
    else
        if *transaction* T_i *has completed*
            inform every participant of the completion of T_i
end

Algorithms Executed by the Participant

After receiving a step, a participant must submit the step to the local scheduler for execution. A step T_{ij} can be submitted to the local scheduler provided $ty(T_{ij})$ is in the successor set of all type of steps of active transactions executing at that site.

Algorithm 5.7.3 *<u>Algorithm for submitting step to local scheduler</u>*

Procedure *Submit-step-to-local* (T_{ij})

begin

  for *each step* T_{mj} *in Active-Set(j)* **do**

    if $ty(T_{ij}) \notin SS(ty(T_{mj}))$

      exit *; /* retry submitting* T_{ij} *later */*

  submit step T_{ij} *to the local scheduler*

  Active-Set(j) $=$ *Active-Set(j)* $\cup \{T_i\}$

end

Once the local scheduler completes a step, the participant informs the coordinator of the outcome of the step.

Algorithm 5.7.4 *<u>Algorithm for sending results to coordinator</u>*

Procedure *Send-Results(result*(T_{ij})*)*

begin

  if T_{ij} *executes successfully*

    send result$(T_{ij}) =$ *success to coordinator*

  else

    send result$(T_{ij}) =$ *failure to coordinator*

end

When a transaction cannot complete successfully, the coordinator sends an abort message to each participant involved in the execution of the transaction. If the participant has already executed or submitted the step S_{ij} for execution, then it adds the compensating step C_{ij} to the set of steps waiting to be executed. If the participant has not yet submitted the step S_{ij} for execution, then it just deletes the step S_{ij} from the set of steps waiting to execute.

Algorithm 5.7.5 *Algorithm for receiving abort message from coordinator*

Procedure *Receive-Abort-Message (S_{ij})*
begin
  if *step S_{ij} has been submitted*
    inserts compensating step C_{ij} into the set of steps waiting to execute
  else
    delete step S_{ij} from the set of steps waiting to execute
end

When a transaction T_i completes, the coordinator sends the participant a completion message. The participant on receiving this message removes T_i from the set of active transactions executing at that site.

Algorithm 5.7.6 *Algorithm for receiving completion message from coordinator*

Procedure *Complete-Trans-Message (T_i)*
begin
  Active-Set(j) $=$ *Active-Set(j)* $-\ T_i$
end

5.7.2 Proof of Correctness

Theorem 5.7.1 All complete histories generated by the above mechanism are stepwise serializable.

Proof:

We need to prove that any history H generated by the mechanism outlined above is conflict equivalent to a correct successor set history.

We construct a history H_s from H as follows. At each site i, we construct a sequence H_i which corresponds to the serialization order of global steps and local transactions of H executing at site i. From the set of sequences H_i we construct a larger sequence H_s as follows: (a) H_s preserves the relative ordering of steps in each H_i, (b) For each pair of global steps T_{ij} and T_{pq} such that the global step T_{ij} is submitted for execution after the execution of the global step T_{pq}, T_{ij} is inserted after T_{pq} in H_s.

We claim that H_s is a correct successor set history. To substantiate this claim we must show two things: (i) H_s is stepwise serial, and (ii) if T_i is incomplete in the prefix of H_s that ends in T_{pj} such that T_{ij} is the last step of T_i that conflicts with and precedes T_{pj} in H, then $ty(T_{pj}) \in SS(ty(T_{ij}))$.

For (i) we need to show that H_s preserves the order of steps of each transaction T_i in H_s. Since the global transaction manager submits the steps for execution in an order consistent with the ordering of steps in a transaction and this ordering is preserved in H_s, H_s is stepwise serial.

For (ii), assume the contrary. Suppose T_i is incomplete in the prefix of history H_s ending in T_{pj}, and T_{ij} is the last step of T_i that conflicts with and precedes T_{pj}, and $ty(T_{pj}) \notin SS(ty(T_{ij}))$. We argue that this case will never arise in our mechanism.

The global transaction manager at site j ensures that $ty(T_{pj})$ is in the successor set of all types of committed steps of active transactions executing at that site, before it dispatches T_{pj} to the local scheduler. Since $ty(T_{pj}) \notin SS(ty(T_{ij}))$, the global transaction manager at site j will not submit T_{pj} until transaction T_i completes and T_{pj} will not appear in the prefix of H_s in which T_i is not complete.

We have thus proved that H_s is a correct successor set history. It remains to be shown that H and H_s are equivalent. H_s and H are composed of the same steps. Since H_s preserves the serialization order of steps and transactions executing at each site, the conflicting steps are ordered the same way in H and H_s. Thus H and H_s are equivalent and H is stepwise serializable.

$\square$

Chapter 6

Our Model for the Multilevel Secure Domain

In this chapter we characterize our model as it applies to multilevel secure database; later in Appendix C (page 146) we show how an example multilevel secure application can be analyzed using the Object Z specification language.

In a system, information flows from one object to another. In a secure database we must ensure that the flow of information does not cause any breach of security, that is, only legal information flow can occur. To prevent direct illegal information flow, the database objects and transactions are assigned security levels and mandatory access control policies govern how transactions can access these objects. Such a database in which objects, transactions are assigned security levels, and mandatory access control policies are enforced is known as a multilevel secure database.

A multilevel secure database is characterized by a security structure that is a partial order, $(\mathbf{C}, <)$ (henceforth called the security poset* structure). $\mathbf{C}$ 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 $C_2 \leq C_1$. We discuss rules preventing direct illegal information flow after describing the objects, integrity constraints, and transactions of a multilevel secure database.

*partially ordered set

Each object in a multilevel secure database is associated with a single security level. As before we have integrity constraints defined on these objects. 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 (single-level integrity constraints), or they may be defined over objects belonging to different security levels (multilevel integrity constraints). A step of level C can only view those integrity constraints which involve objects at level C or below. Consequently, only the highest step, if present, can view all the integrity constraints of the database.

The database objects are accessed using transactions. A transaction consists of a set of operations. Each operation in a transaction is also associated with a security level. A transaction is single-level if all its operations have the same security level, otherwise it is multilevel. A single-level transaction can be viewed as a special case of multilevel transactions and from now on in this chapter we use the term 'transaction' to mean a multilevel transaction.

Information flow from the dominated level to the dominating level is permitted; all other information flows are considered to be illegal. Direct illegal information flow is prevented by enforcing the mandatory access control policy derived from the Bell-Lapadula model [San93]. This policy comprises of two parts. (a) Simple Security Property : an operation can read a database object only if its level dominates the level of the object it reads. (b) Restricted *-Property : an operation can write a database object only if its level is dominated by the level of the object it writes.

Although the mandatory access control policies prevent direct illegal information flow, illegal information flow may occur indirectly. For instance a high process (a process at the dominating level) can signal information to another process by modulating its own use of system resources (example, CPU time) in such a way that this manipulation affects the real response time observed by a low process (a process at the dominated level). The illegal information flow described above is known as a timing channel. There are other indirect means by which information can be transmitted illegally. Our objective is to prevent such indirect communication, that is, we need to ensure that when transactions execute no operation at the dominated level can be influenced by any operation at the dominating level.

6.1 Decomposition of Multilevel Transactions

In this section, we describe in details the decomposition process of a multilevel transaction. The decomposition process is complicated for security reasons – no information must be transmitted from a dominating level to a dominated level by virtue of executing the multilevel transaction.

In Chapter 3 we stated that 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 usually involve reading and writing objects. Thus we define multilevel transactions as follows:

Definition 6.1.1 **[Multilevel Transaction]** A multilevel transaction T_i is a set of read and write operations in which conflicting operations are ordered by the relation $\prec_i$.

A read or write operation in a multilevel transaction is associated with a security level. To prevent direct illegal information flow, the security level of a write operation must be equal to the level of the object it writes and the security level of a read operation must dominate the level of the object it reads. The level of each operation in a multilevel transaction must lie within the range of security levels associated with the multilevel transaction. 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 limit the size of trusted code, we follow other authors [CM92, CJ93] and decompose a multilevel transaction into single-level steps[†]. To keep our model simple, we assume that a transaction has at most one step at any security level. Each write operation in a multilevel transaction is associated with exactly one step, but a single read operation in the multilevel transaction may appear in multiple steps. Let the function L map database objects and steps to security levels. We require a step S_{ij} of multilevel transaction T_i to obey the simple security property and the restricted $\star$-property of the Bell-Lapadula model [BL75]:

1. A step S_{ij} with $L(S_{ij}) = C$ may read a database object x if $L(x) \leq C$.
2. A step 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 6.1.2 [Transaction Decomposition] A multilevel transaction T_i is *decomposed* into a set of steps $\{S_{i1}, S_{i2}, \ldots, S_{in}\}$ such that

1. conflicting operations in step S_{ij} are ordered by the relation $\prec_{ij}$,
2. orderings of conflicting operations specified in a step honor the orderings of conflicting operations specified in the transaction, that is, $\prec_{ij} \subseteq \prec_i$, and

[†]To make this chapter consistent with the rest of the thesis we use the term 'step' to refer to a decomposed unit and not use the term 'section' as used in the security literature.

3. each write operation in T_i appears in exactly one step and each read operation in T_i appears in at least one step.

To avoid timing channels, it is necessary to order the steps in a way that is consistent with the security poset structure (low to high order). However given a transaction decomposed according to Definition 6.1.2 it may not always be possible to execute the steps in a low to high order, and at the same time respect the ordering relation $\prec_i$.

Note that this problem has been solved by Costich and Jajodia [CJ93], and Smith et al. [SBJN96]. The solution appropriate for our model, derived from that of Smith's [SBJN96], is presented next.

6.1.1 Ordering Steps in a Transaction

Before we show how to order the steps in a transaction according to the security poset structure, we need the notion of dependencies from the work done by Blaustein et al. [BJMN93]. A *dependency* exists between two steps if they contain conflicting operations. There can be two types of dependencies between steps of a transaction, depending on the ordering of operations in the underlying multilevel transaction.

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

We adopt a caching technique [SBJN96] as the mechanism to handle the dependencies in a correct manner. The cache is local to the transaction; it is visible only to steps of a single transaction. In accordance with security requirements, a high step can read the cached values written by the low step but not vice-versa.

Consider a L-H dependency in transaction T_i: $w_i[x] \prec_i r_i[x]$ where $w_i[x]$ is an operation in the low step and $r_i[x]$ is an operation in the high step. The low step writes the new value of x in the cache. Subsequently, the high step reads this new value from the cache.

Consider a H-L dependency in transaction T_j: $r_j[x] \prec_j w_j[x]$ where $r_j[x]$ is an operation in the high step and $w_j[x]$ is an operation in the low step. When the low step executes, it writes the old value and new value of x in the cache. The database read operation $r_j[x]$ in the high step is converted to a cache read; the high step reads the old value of x from the cache. Note that in this case although the high step executes after the low step, it will still read the correct value of x. This example helps to illustrate how the caching technique converts a H-L dependency to a L-H dependency.

The caching technique can thus eliminate all H-L dependencies. Once all the H-L dependencies are eliminated, the only dependencies that exist between the steps of a transaction are the L-H dependencies. With only L-H dependencies remaining, it is possible to order the steps of a transaction low first.

Definition 6.1.3 [Partial Order $\ll_i$ On Steps In Transaction T_i] Let $\ll_i$ be a partial order defined on the steps of transaction T_i. For any two steps S_{ij},S_{ik} of transaction T_i which are related by the partial order $\ll_i$, $S_{ij} \ll_i S_{ik}$, if all operations of step 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 steps are ordered according to the security poset structure.

Definition 6.1.4 [**Canonical Form**] A transaction T_i is in *canonical form* if for all j, k, $S_{ij} \ll_i S_{ik}$ iff $L(S_{ij}) < L(S_{ik})$.

The ordering of steps of a transaction according to the security poset structure is important for security reasons. For example, if the steps of a transaction are not ordered then a low process may observe the time taken for a high step to execute and infer information. 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 (algorithm in which the dominated steps of a transaction are executed before the dominating steps).

After describing the decomposition process of multilevel transactions we now describe how the integrity constraints can be generalized in a secure environment.

6.2 Generalizing the Multilevel Integrity Constraints

In the secure domain, the multilevel integrity constraints must be generalized such that the execution of each step satisfies the generalized integrity constraints. We use auxiliary variables to generalize the original integrity constraints. To prevent illegal information flow we must enforce the following restrictions.

1. An auxiliary variable is associated with only one security level.

2. Bell-Lapadula rules are followed in accessing these auxiliary variables.

3. An auxiliary variable whose level is dominated by C cannot be used to generalize an integrity constraint which involves database objects at level C or below.

6.3 Compensating Steps

Compensation is problematic from the security point of view. Compensating steps are, therefore, not specified for the decompositions of multilevel transactions. The example below illustrates why compensation is undesirable in a secure environment.

Example 6.3.1 *Consider a transaction T_i decomposed into steps $< S_{i1}, S_{i2} >$ where $L(S_{i1}) = Low$ and $L(S_{i2}) = High$. Suppose step S_{i1} commits, but step S_{i2} aborts. To neutralize the effect of S_{i1}, a compensating step C_{i1} must execute. Since C_{i1} semantically undoes the actions of S_{i1}, level of C_{i1} must equal that of S_{i1} which in this case is Low. Assume that there resides a pair of communicating processes at the two levels. The High process can selectively abort and commit S_{i2} to pass one bit of information. A Low process can observe whether or not a compensating step is initiated and be able to infer the information sent by the High process.*

6.4 Semantic Histories

Since we no longer have compensating steps, the notion of stepwise serial history given in Definition 3.4.1 must be changed for this context.

Definition 6.4.1 [**Stepwise Serial One Version History**] A stepwise 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 steps with the ordering relation $\ll_H$ such that,

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

2. for any two steps S_{ij}, S_{ik} of some $T_i \in \mathbf{T}$, $S_{ij} \ll_H S_{ik}$ in H if $S_{ik} \not\ll_i S_{ij}$ in canonical form of T_i,

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

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

The notion of complete execution given in Definition 3.4.2 must be modified in the absence of compensation. Note that, in this case, the only possible execution sequence of a transaction is the successful execution sequence.

Definition 6.4.2 [Complete Execution] An execution of a transaction $T_i = < S_{i1}, S_{i2}, \ldots, S_{in} >$ in a stepwise serial one version history H is a *complete* execution if all n steps of T_i appear in H.

A semantic history, as before, captures the states associated with the execution of each step in the corresponding stepwise serial one version history. The correctness notion of semantic histories, given in Definitions 3.4.5 and 3.4.6, is applicable for this model as well.

6.5 Properties of the Decomposition

The composition property, sensitive transaction isolation property, consistent execution property are defined as before.

The absence of compensation reduces the semantic atomicity property to the successful execution property. This is because from Definition 6.4.2 a transaction T_i has only one possible execution sequence which is the successful execution sequence and the only way in which transaction T_i can complete is successfully.

It is not hard to find examples not having the successful execution property. Please refer to page 152 in Appendix C to see how a slight modification of the requirements causes the Mission Database example to lose the successful execution property. Below we give a more general completeness property which we call the complete execution property.

[**Complete Execution Property**] Every correct partial semantic history H is a prefix of some correct complete semantic history.

The complete execution property 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 step is being triggered to complete a high step of any transaction. Note that, unlike the successful execution property which is optional, the complete execution is a necessary property.

The proof of the properties for a Mission Database, an example multilevel application, appears in Appendix C.

6.6 Successor Sets

In the previous chapters we discussed the use of successor sets to specify allowable versus non-allowable interleavings. In the context of a multilevel secure database, however, the use of successor sets may be exploited for the purpose of illegal information flow. We show this with an example.

Example 6.6.1 *Consider the following scenario. We have two transactions T_1 and T_2: T_1 is composed of two steps, S_{11} and S_{12}, where $L(S_{11}) = Low$ and $L(S_{12}) = High$. T_2 consists of the single step S_{21} where $L(S_{21}) = Low$. Suppose $ty(S_{21}) \notin SS(ty(S_{11}))$ and we have the following correct semantic history: $H =< S_{11} >$. At this point step S_{21} wishes to execute. Now S_{21} will not be allowed to execute until step S_{12} completes because $ty(S_{21}) \notin SS(ty(S_{11}))$.*

In the following we show how one bit of information can be passed on from the High level to the Low level using the successor set mechanism. To illegally transfer this information there must be two cooperating processes executing at the two levels High and Low. There is an understanding between the two processes that if S_{21} experiences a long delay before it can be executed, then the High process wants to send bit 1, otherwise it wants to send bit 0. The High process can delay the execution of S_{12} (for example, by locking the resources required by S_{12}). Because of successor set restrictions step S_{21} cannot execute until step S_{12} completes. The Low process can monitor the delay experienced by S_{21} and infer the one bit of information sent by the High process.

Since successor sets can be exploited for illegal information flow unless additional measures are taken, we do not use successor sets for controlling the interleavings of multilevel transactions. Thus all interleavings generated from the decomposed specifications must be allowable. Alternatively, we can view this as the successor set of each type of step containing all the types of steps. With such a successor set description, a correct successor set history, as defined in 3.6.2, reduces to a correct semantic history.

In a correct semantic history the steps of a transaction execute serially. To improve the performance the steps of transactions must be executed concurrently. The next section describes the notion of correctness for concurrent executions of multilevel transactions.

6.7 Correct Concurrent Executions

Multilevel secure databases often support some form of multiversioning to prevent starvation of transactions executing at the dominating level. For this reason, we need to develop a new notion of correct concurrent execution.

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 step 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 6.7.1 [**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 steps, is a partial order with ordering relation $\prec_H$ 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 $p_{ij} \prec_{ij} q_{ij}$, then $h(p_{ij}) \prec_H h(q_{ij})$;

3. if $h(r_{ij}[x]) = r_{ij}[x_{kl}]$, then $w_{kl}[x_{kl}] \prec_H r_{ij}[x_{kl}]$; and

4. if $w_{ij}[x] \prec_{ij} r_{ij}[x]$, then $h(r_{ij}[x]) = r_{ij}[x_{ij}]$.

Condition (1) states that the multiversion history contains the translations of the operations in each step. Condition (2) states that the multiversion history preserves all the operation orderings specified by the steps. Condition (3) states that a step cannot read a version before it has been created. Condition (4) states that if a step 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 stepwise serial one version histories. For this we need the notion of equivalence of histories; this definition is similar to the one in [BHG87]. Before defining equivalence we need the notion of reads from relationship.

Definition 6.7.2 [S_{ij} **reads** x **from** S_{kl}] For a one version history, S_{ij} reads x from S_{kl} if $w_{kl}[x] \prec_H r_{ij}[x]$ and there is no $w_{mn}[x]$ such that $w_{kl}[x] \prec_H w_{mn}[x] \prec_H w_{ij}[x]$. For a multiversion history, S_{ij} reads x from S_{kl} if the operation $r_{ij}[x_{kl}]$ is present in the history.

Definition 6.7.3 [**Equivalence of Histories**] Two histories H and H' are said to be equivalent, if they are defined over the same steps 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 stepwise serial.

Definition 6.7.4 [Stepwise Serial Multiversion History] A multiversion history is stepwise serial if for every two steps 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 all stepwise serial multiversion histories are not equivalent to stepwise serial one version histories; the following definition characterizes those stepwise serial multiversion histories which are equivalent to stepwise serial one version histories.

Definition 6.7.5 [One-Copy Stepwise Serial Multiversion History] A stepwise serial multiversion history H is one-copy stepwise 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 step 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 stepwise serializable multiversion history in which steps may not be executed serially, but the effect of the history is the same as a one-copy stepwise serial multiversion history.

Definition 6.7.6 [One-Copy Stepwise Serializable Multiversion History] A one-copy stepwise serializable multiversion history is one that is equivalent to a one-copy stepwise serial multiversion history.

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

6.8 Concurrency Control Mechanism

In this section we propose a concurrency control mechanism for a kernelized architecture which is based on the algorithm in [CJ93]. The scheduler is divided into two parts: a trusted global scheduler which controls the proper sequencing of steps of a transaction, and untrusted local schedulers at each security level which are responsible for scheduling the operations of a step.

We make the following assumptions:

1. The global scheduler is trusted and centralized.
2. The global scheduler receives the information about the relative ordering of the steps of a transaction.
3. The read sets, that is, the set of database objects read by a step, is known in advance.
4. The write sets, that is, the set of database objects written by the step, is known in advance.
5. The algorithms given below execute atomically.

For the purpose of this section we define a step as a sequence of read and write operations followed by a commit or an abort operation,

$$S_{ij} =< O_{ij}(x_1), O_{ij}(x_2), \ldots, O_{ij}(x_n), E_{ij} >$$

where $O_{ij}(x)$ is either $R_{ij}(x)$ or $W_{ij}(x)$ and E_{ij} is either C_{ij} or A_{ij}.

6.8.1 Algorithms

Algorithms Executed by the Global Scheduler

The global scheduler is ready to dispatch a step S_{ij} to the local scheduler when all the steps 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 $L(S_{ij})$.

Algorithm 6.8.1 *Algorithm for Dispatching Steps to Local Scheduler*

Procedure *Dispatch-step-to-local (S_{ij})*

begin

  for *each step S_{im} in T_i* **do**

    if $L(S_{im}) < L(S_{ij})$ *and step S_{im} has not committed*

      exit;   /* S_{ij} *must retry later* */

  generate $TS(S_{ij})$

  send S_{ij}, $TS(S_{ij})$, $RS(S_{ij})$, $WS(S_{ij})$ to scheduler at level $L(S_{ij})$

end

After a local scheduler completes the execution of some step S_{ij}, it informs the global scheduler of the outcome. The global scheduler records the outcome of step S_{ij} – this information is required by the global scheduler for dispatching the other steps of T_i. If some step cannot be successfully executed, the user is informed of this unsuccessful execution.

Algorithm 6.8.2 *Algorithm for Receiving Results*

Procedure *Receive-execution-result (result(S_{ij}))*

begin

record the status of execution

if $result(S_{ij}) = abort$

inform user of unsuccessful execution of S_{ij}

end

Algorithms Executed by the Local Scheduler

The local scheduler does not execute step S_{ij} until all steps, 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 step S_{kl} and S_{kl} is the step with the highest timestamp not greater than $ts(S_{ij})$ which writes into any version of x.

Algorithm 6.8.3 *Algorithm for Read*

Procedure *Process-read* $(R_{ij}(x))$

begin

for *each step* S_{kl} *which is waiting or executing and* $L(S_{kl}) \leq L(S_{ij})$ **do**

if $WS(S_{kl}) \cap RS(S_{ij}) \neq \varnothing$

exit; /* Step S_{ij} must retry later */

if $TS(S_{mn}) < TS(S_{ij})$ *and* $x \in WS(S_{mn})$

$readFrom = S_{mn}$

for *each* S_{pq} *such that* $x \in WS(S_{pq})$ **do**

if $TS(readFrom) < TS(S_{pq}) < TS(S_{ij})$

$readFrom = x_{pq}$

$R_{ij}(x)$ *translated into* $R_{ij}(x_{readFrom})$

Accept $R_{ij}(x_{readFrom})$

end

As before the local scheduler first checks whether the operations of step S_{ij} can execute. During the write operation of data object x, the appropriate version is created.

Algorithm 6.8.4 *Algorithm for Write*

Procedure *Process-write* $(W_{ij}(x))$

begin

for *each step* S_{kl} *which is waiting or executing and* $L(S_{kl}) < L(S_{ij})$ **do**

if $WS(S_{kl}) \cap RS(S_{ij}) \neq \varnothing$

exit; /* *Step* S_{ij} *must retry later* */

$W_{ij}(x)$ *is translated into* $W_{ij}(x_{ij})$

Accept $W_{ij}(x_{ij})$

end

A step commits when all the operations of a step are complete. In addition to the functions performed during commit, our algorithm also requires the local scheduler to notify the global scheduler of the outcome.

Algorithm 6.8.5 *Algorithm for Step Commit*

Procedure *Process-stepcommit*(C_{ij})

begin

send commit message to global scheduler

end

A step may not always complete successfully; it may have to abort. Traditional recovery for aborted transactions, such as undo, (which involves destroying the versions created by the aborted steps) is required for the aborted step, but details are omitted.

Algorithm 6.8.6 *Algorithm for Step Abort*

Procedure *Process-stepabort(A_{ij})*

begin

send abort message to global scheduler

end

6.8.2 Proof of Correctness

Theorem 6.8.1 A history H generated by the concurrency control algorithm is a one-copy stepwise serializable history.

Proof: Apply the following transformation on the history H: arrange the operations to get a stepwise serial history H_s. The order of steps in H_s must be the same as the timestamp ordering. Now we will show that H_s is a one-copy stepwise 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 step 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 step 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 step and dispatches the step to the local scheduler only after the dominated steps of the same transaction have committed. Hence in a transaction, a step at a dominating level has a higher timestamp than a step at the dominated level. So in H_s, the step at the dominated level will appear before the step at the dominating level. Thus H_s preserves the canonical ordering of a transaction. Since H_s satisfies (i) and (ii), it is a one-copy stepwise serial history. Since H and H_s are composed of the same operations they are equivalent; therefore H is a one-copy stepwise serializable history. □

6.8.3 Proof of Security

Theorem 6.8.2 Our algorithm is secure, that is, (i) there is no illegal information flow from the dominating to the dominated levels, and (ii) our algorithm does not introduce any covert channels.

Proof: The multilevel transactions are decomposed into single level steps. A step is allowed to read objects at dominated levels and write objects at its 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 step at the dominated level is aborted because of some step at the dominating level. First we show that no operation in a dominated step is delayed by any operation in the dominating step. Let S_{ij} be the dominated step and S_{kl} be the dominating step. 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 step of a transaction is initiated only after the dominated step 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 $ts(S_{kl}) < ts(S_{ij})$, step 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 step because of some operation in the dominating step. Let S_{ij} be the dominated step and S_{kl} be the dominating step. For $i = k$, the proof is trivial, as the dominated steps of a transaction are committed before the initiation of the dominating steps. For $i \neq k$, we must show that S_{ij} is not aborted because of S_{kl}. For this let us consider the step aborts that may occur for concurrency control reasons. Note that in timestamp based algorithm if a step S_{ij} with a smaller timestamp tries to write a data item after another step S_{kl} with a larger timestamp has already read the data item, step S_{ij} will be aborted. In our algorithm this possibility is avoided by requiring steps with later timestamps to wait until all steps, at dominated levels whose write sets have a non-null intersection with the read set of the dominating step, 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. $\square$

Chapter 7

Automated Analysis

One disadvantage of semantic-based models is that they require analysis of applications in order to provide advantages in terms of performance, autonomy or security. Specifically, the necessary properties, stated in the previous three chapters, must be proved for applications that are to be processed using our model. Verifying the application for the satisfaction of these properties puts an extra overhead on the application developer; this overhead involves formulating and discharging the proof obligations associated with the properties. The application developer may choose the appropriate level of rigor for discharging the proof obligations; for critical applications formal verification of the properties may be required, and for others an intuitive argument might suffice. However, for real world complex applications proving the properties using hand analysis may be tedious and error-prone. Towards this end, we show the extent to which the two standard approaches of automated software verification can be used to ensure the satisfaction of the necessary properties.

First we present a motivating example application and describe how a transaction in this application is decomposed. Later we describe how this decomposition can be verified by the theorem proving and model checking approaches.

7.1 Informal Description of an Example Decomposition

We describe a very simple example of a hotel database. The hotel application is an example application in the long duration domain and the formal description using the specification language Object Z appears in Appendix A.

7.1.1 Hotel Example - Before the Decomposition

The hotel database has a set of rooms. The total number of rooms in the hotel is stored in object *total* and *res* stores the number of reservations. The object ST stores the status information of each room in the hotel; the status of each room is either *Taken* or *Available* depending on whether or not the room is occupied by any guest. The object RM maintains information about guest to room assignments.

The hotel database has the following integrity constraints:
(i) each room can have only one status – *Taken* or *Available*,
(ii) each guest can be assigned to at most one room,
(iii) each room can be assigned to at most one guest,
(iv) the number of reservations equals the number of rooms occupied by guests, and
(v) the set of rooms with status *Taken* equals the set of rooms assigned to guests.

There are three types of transactions in the hotel database – *Reserve*, *Cancel* and *Report*.* The *Reserve* transaction reserves a room for a guest who does not have any existing reservations. The input to the *Reserve* transaction is *g_inp* (the guest for whom the room is being reserved) and *r_inp* (the room requested by the guest). The precondition to the *Reserve* transaction is that the number of reservations (*res*) must be less than the total number of rooms in the hotel (*total*), the room requested

*Note that a real world hotel application will have many more types of transactions like $Check-in$ and $Check-out$; but to keep the example simple we show only three types of transactions.

(*r_inp*) is *Available*, and the guest (*g_inp*) does not have any pre-existing reservation. The postcondition of *Reserve* increments the number of reservations (*res*), changes the status of selected room (*r_inp*) to *Taken* and assigns the room (*r_inp*) to the guest (*g_inp*). The *Cancel* transaction cancels the reservation for a guest *g_inp*. The input to the *Cancel* transaction is *g_inp*. The precondition of the *Cancel* transaction is that the guest whose reservation is being canceled (*g_inp*) has a room assigned to him. The postcondition of *Cancel* is that the status of the room assigned to *g_inp* is changed to *Available* and the *g_inp* is deassigned from this room. The *Report* transaction prints the information about the status of each room and the guest to room assignments. The *Report* transaction has no preconditions and produces as outputs the contents of ST and RM.

7.1.2 Hotel Example - After the Decomposition

The *Reserve* transaction is decomposed into three steps, namely, $R1$, $R2$ and $R3$. As stated in Section 3.2, the decomposition necessitates the generalization of integrity constraints. For this purpose we introduce two auxiliary variables *underway* and *acquired*; *underway* gives the number of reservations in progress and *acquired* gives the set of rooms whose status have been changed but which have not yet been assigned to guests.

The integrity constraints (i), (ii) and (iii), specified in Section 7.1.1 are not generalized. The integrity constraints (iv) and (v) are generalized as follows: (iv) the number of reservations (*res*) equals the number of rooms occupied by guests plus the number of reservations in progress (*underway*), (v) the set of rooms with status *Taken* equals the set of rooms assigned to guests and the rooms in the set *acquired*. In addition, there is another constraint involving the auxiliary variable *acquired*: (vi) the rooms in the set *acquired* are not the ones assigned to guests.

The step $R1$ updates the number of reservations. The precondition of $R1$ is that *res* must be less than *total*. The postcondition of $R1$ increments *res* and *underway*. The step $R2$ is responsible for changing the status of the room selected by the customer. $R2$ requires as input *r_inp* - the room number of customer's preference. The precondition of $R2$ is that the room *r_inp* must be available and *underway* must be positive (to ensure that $R1$ has been executed before $R2$ executes). The postcondition of $R2$ changes the status of *r_inp* to *Taken* and inserts *r_inp* in the set *acquired*. The step $R3$ assigns the room picked by the customer to the guest. The step $R3$ requires *r_inp* and *g_inp* as its input. The precondition of $R3$ is that the guest *g_inp* must not have any existing reservations, *underway* is positive and *r_inp* is in the set *acquired*. The postcondition of $R3$ assigns *r_inp* to the guest *g_inp*, decrements *underway*, removes *r_inp* from the set *acquired*.

The decomposition also necessitates the specification of compensating steps. The compensating step $CompR1$, $CompR2$ compensates the effects of step $R1$, $R2$ respectively. The precondition of $CompR1$ is that *res* and *underway* must be positive. The postcondition of $CompR1$ decrements *res* and *underway*. The precondition of $CompR2$ checks that *r_inp* must be in the set *acquired*. The postcondition of $CompR2$ removes *r_inp* from the set *acquired* and changes the status of *r_inp* to *Available*.

The *Cancel* and *Report* transactions are not decomposed into steps and are specified as in Section 7.1.1.

7.2 Theorem Proving

Theorem proving has long been used as an automated verification technique. It requires the system to be verified to be represented in some logic and the properties also expressed in the same logic. Theorem proving involves deducing the property from the axioms and the rules of inference. Once a theorem pertaining to some property is proved about a particular system, we have assurance that the property does indeed hold in that system.

7.2.1 The B-Toolkit

As a feasibility study of the theorem proving approach we use the B-Toolkit [Sor94] to verify semantic-based decompositions. The B-Toolkit is the tool support for the B-Method [Abr96, Wor96], which provides a formal approach for developing high integrity software. The B-Toolkit is commercially available and has been used in the development of safety critical applications.

The input to the B-Toolkit is an AMN (Abstract Machine Notation) specification. An AMN specification consists of two parts, namely a static part and a dynamic part. The static part contains a declaration of mathematical objects needed to model the state. These objects include basic sets, variables and constants. The static part also includes invariants which constrain the values of the variables. The dynamic part is composed of the initialization operation and the operations defined on the state variables. Once a system has been specified, the consistency of the system can be checked by ensuring that the invariants are satisfied by the values of the variables which are changed according to the operations specified in the dynamic part.

Once the AMN specification has been specified, the Proof Obligation Generator in the B-Toolkit is responsible for generating all the proof obligations associated with invariant maintenance. These proof obligations can be discharged using the AutoProver (the automated theorem prover) or the InterProver (the interactive theorem prover) of the B-Toolkit. For simple proof obligations, the AutoProver might suffice; for others the user may have to supply user theories through the InterProver to aid in the discharge of proof obligations.

7.2.2 AMN Specification of the Hotel Example

The B-Toolkit accepts AMN specifications. In this section we describe the the AMN specification of the Hotel Example. The complete specification of the Hotel Example appears in Appendix D.

Sets

Each variable specified in the static part should belong to some basic set. Basic sets can be predefined (for example, set of natural numbers) or they can be defined by the user (for example, enumerated sets). In the Hotel Example we define the set $GUEST$ and $ROOM$ which enumerate all possible guests and all possible rooms respectively. We define an enumerated type $STATUS$ which lists the possible values that a status of a room may take, which, in this case, is *Available* and *Taken*.

Constants

A database object may be a constant. The hotel example has one constant, namely, *total*.

Properties

The values of the constants or any other constraints on constants are specified as properties of the specification. For example, if the number of rooms in the hotel equals 100, we specify $total = 100$.

Variables

The variables in the hotel example are *res*, *underway*, *acquired*, RM and ST.

Invariant

The invariant is a logical property that constrains the values of the variables. Moreover, it should contain enough information to relate each variable to its basic set. The invariant in the AMN specification is the conjunction of the generalized integrity constraints and the type information of the variables. Please refer to Appendix D to see how the invariant is specified for the Hotel Example.

In the following we describe how the generalized integrity constraints of the Hotel Example stated in Section 7.1.2 are mapped to the corresponding AMN specification.

(i) Each room has one status is specified by denoting ST as a partial function mapping rooms to their status: $ST : ROOM \nrightarrow STATUS$

(ii) each guest can be assigned to at most one room and (iii) each room can be occupied by at most one guest is depicted by describing RM as a partial injective function as shown below: $RM : GUEST \rightarrowtail\!\!\!\!\!\!+\; ROOM$

(iv) the number of reservations equals the number of rooms occupied by guests and the number of reservations in progress: $res = card(RM) + underway$

(v) the set of rooms with status *Taken* equals the set of rooms assigned to guests and the set of rooms which have status *Taken* but which have not yet been assigned to guests: $\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}(RM) \cup acquired$

(vi) the rooms in the set *acquired* are not the ones assigned to guests: $\mathrm{ran}(RM) \cap acquired = \varnothing$.

Initialization

The initialization operation consists of establishing the initial state. The initialization operation, as well as other operations, are defined using the notion of substitution.

Each substitution S is defined as a predicate transformer which transforms a postcondition R into the precondition for S to establish R. The notation ':=' represents substitution in AMN. When multiple variables are initialized, parallel substitution must be used.

In the hotel example, the value of *res*, *underway* are initialized to 0 and *acquired*, *RM* are initialized to the empty set. Initially all rooms have status *Available*, so ST is initialized appropriately. In the AMN specification these are described as follows:

$res, underway, acquired, RM, ST := 0, 0, \varnothing, \varnothing, ROOM \times \{Available\}$

Operations

The operations in an AMN specification are defined as *pre-conditioned substitution*, which means the substitution is carried out only if the predicate denoting the precondition of the operation holds. If the operations are carried out when the predicates are not satisfied, anything might happen. The substitution does not refer to variables which do not change in the course of the operation. When multiple variables are updated, parallel substitution must be used. The individual substitutions in a parallel substitution are separated by the '||' operator. The substitutions comprise the *body* of the operation [SH94]. The predicate that holds after the substitutions specified in the body have been carried out is referred to as the postcondition of the operation.

Corresponding to each step and compensating step in the Hotel Example we specify an operation. The precondition of the operation is a conjunction of the preconditions of the step and the type information of the input variables. The body of the operation consists of substitutions. The body of the operation is derived from the postcondition of the step. For each variable updated in the postcondition, a substitution is specified describing how the variable is updated by the operation.

For example step $R2$ is specified as follows:

$$
\begin{array}{l}
R2\ (\ room\_inp\ )\ \widehat{=} \\
\quad \textbf{PRE} \\
\qquad room\_inp \in ROOM \wedge underway > 0 \wedge ST(room\_inp) = Available \\
\quad \textbf{THEN} \\
\qquad ST := ST \lhd\!\!\!- \{room\_inp \mapsto Taken\} \ || \\
\qquad acquired := acquired \cup \{room\_inp\} \\
\quad \textbf{END}
\end{array}
$$

7.2.3 Analysis and Conclusions

The AMN specification allows one to specify only the invariant properties of a system. Consequently, the B-Toolkit allowed us to specify and verify only the generalized integrity constraints. The Proof Obligation Generator of the B-Toolkit generated 35 proof obligations. The AutoProver was able to discharge 24 proof obligations. The remaining 11 proof obligations could be discharged with the help of user theories given to the InterProver. The discharging of proof obligations associated with the satisfaction of generalized integrity constraints were, therefore, not completely automated.

The rationale for using the B-Toolkit over existing theorem provers such as the PVS [ORS92] is as follows. The B-Toolkit provides support for the entire software development lifecycle. In addition to analyzing the properties, the tools provided by the B-Toolkit can be used to check other consistency properties, animate the AMN specification to help the developer decide whether the original and the decomposed specification conforms to the requirements, and formally refine the AMN specification to executable code in C. In brief, it is advantageous for our analysis to be part of an integrated tool set for software development.

The B-Method, in its present form, does not have temporal support. Consequently, the semantic correctness properties could not be specified or verified. Note that other researchers [LBS96] have also described these shortcomings which make the B-Method inadequate in representing the safety properties of chemical processing system. The authors [LBS96] propose the extensions in B – for example, thread specification, trace and history specifications – which are necessary to model the behavior of reactive systems. The addition of the following features will make it more amenable to verification of semantic based decompositions: (i) temporal support and a trace-based semantics that allows one to reason about sequence of operations, (ii) history specifications that will allow one to specify the restrictions on the execution of operations.

Theorem proving, in general, is technically challenging. It is worth noting a standard property of theorem proving. Specifically, if a proof obligation remains unproved, then there are two possibilities – the 'theorem' may not be a theorem at all, in which case an error has been found, or a proof has simply not yet been identified. In the second case the user may have to supply theories in order to aid the theorem prover. This in turn implies that both domain expertise and theorem proving expertise is necessary to ascertain the cause of the failure in discharging a proof obligation.

7.3 Model Checking

Theorem proving, the traditional approach to automated software verification, is tedious, difficult and time consuming; consequently, even with existing theorem provers like the PVS [ORS92] only small applications can be verified. This motivates us to look at an alternative automated approach, known as model checking, to get assurance for the satisfaction of the properties. Model checking has been used successfully to verify hardware circuits; recently researchers [AG93, WVF96] have advocated 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 all 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 this section we describe how we developed a finite model of the hotel application which we verified using the SMV model checker [McM92].

7.3.1 The SMV Symbolic Model Checker

The input to the symbolic model checker is an 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.

7.3.2 Finite State Abstraction of the Hotel Example

In the following paragraphs we describe a very simple finite model of the hotel example. Appendix E describes the input and the output produced by the SMV model checker.

State Variables

Our model hotel has three rooms. The room ids in the hotel are specified as integers. The room id which is input to the *Reserve* transaction, denoted by *r_inp*, can take on any value in the range 0 to 3. The room id which is passed as a parameter from $R2$ to $R3$ (or $CompR2$) is denoted by *pr_inp* and has the range from 0 to 3. The variable *res* is specified as an integer in the range 0 to 3. Similarly *underway* is also specified as an integer with the same range. The variable *acquired* is specified as a fixed array; $acquired[x]$ indicates whether or not the room x is in the set *acquired*. If $acquired[x]$ is 1, it means that the room x is in the set *acquired*, otherwise it is not. The variable ST gives the status information of each room. ST is specified as a fixed array. $ST[x]$ gives the status of room x – the status of room x can take on two values: 0 (indicating that room x is *Available*) or 1 (indicating that room x is taken). There is a finite set of guests who make reservations to the hotel. This finite set is made up of guests a, b, c and d. The guest id which is an input to the *Reserve* or *Cancel* transaction is denoted by *g_inp* and can take on values in the set $\{a, b, c, d\}$. The variable *RMI* which stores the guest information is also specified as a fixed array. $RMI[x]$ denotes the guest assigned to room x. When no guests are assigned to room x, the value of $RMI[x]$ is n.

We have one variable *input* which enumerates all possible types of steps and compensating steps in the hotel example. Thus for the hotel example, the variable *input* can take on values $R1$, $R2$, $R3$, $CompR1$, $CompR2$, $CancelD$ and $ReportD$.

Our model limits the number of *Reserve* transactions that are executed. The constant $maxR$ specifies the maximum number of *Reserve* transactions allowed. To verify the consistent execution and semantic atomicity property we need to keep track of the number of times the steps and compensating steps of the *Reserve* transaction has been executed. For this purpose we specify the counter variables $r1$, $r2$, $r3$, $compr1$, $compr2$. All these variables are integers and the maximum value of each variable depends on the maximum number of *Reserve* transactions permitted.

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. Initially *res*, *underway* and all the counter variables are set to 0. Since all rooms are available in the initial state, $ST[x]$ is assigned the value 0. In the absence of guests in each room x, $RMI[x]$ is initialized to n. 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 $RMI[1]$ is given below.

$init(RMI[1])\ :=\ n;$
$next(RMI[1])\ :=\ case$
$(input = R3)\ \&\ (r3 < maxR)\ \&\ (underway > 0)\ \&\ (pr\_inp = 1)\ \&\ acquired[1]\ \&$
$!(RMI[1] = g\_inp)\ \&\ !(RMI[2] = g\_inp)\ \&\ !(RMI[3] = g\_inp)\ :\ g\_inp;$
$(input = CancelD)\ \&\ (RMI[1] = g\_inp)\ :\ n;$
$1\ :\ RMI[1];$
$esac;$

7.3.3 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 specified in Section 7.1.2 is mapped to the corresponding CTL formula.

(i) Each room can have only one status: the definition of ST as a fixed array ensures this constraint; the variable $ST[x]$ denoting the status of room allows $ST[x]$ to take only one value: 0 or 1.

(ii) Each room can be assigned to at most one guest: the definition of *RMI* as a fixed array takes care of this constraint; $RMI[x]$ can take only one value which means that at most one guest can be assigned to a room.

(iii) Each guest can have at most one reservation: this is modeled by the 12 CTL formulae given below. Recall that a CTL formula with the prefix AG means 'for all paths, for all states'. The first formula says that it is always the case that if room 1 is occupied by guest a , then guest a does not occupy any of the other rooms. The other formulae can be interpreted similarly.

$$
\begin{array}{lll}
\mathit{SPEC}\ AG(\mathit{RMI}[1] & = & a -> !(\mathit{RMI}[2] = a)\ \&\ !(\mathit{RMI}[3] = a)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[1] & = & b -> !(\mathit{RMI}[2] = b)\ \&\ !(\mathit{RMI}[3] = b)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[1] & = & c -> !(\mathit{RMI}[2] = c)\ \&\ !(\mathit{RMI}[3] = c)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[1] & = & d -> !(\mathit{RMI}[2] = d)\ \&\ !(\mathit{RMI}[3] = d)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[2] & = & a -> !(\mathit{RMI}[1] = a)\ \&\ !(\mathit{RMI}[3] = a)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[2] & = & b -> !(\mathit{RMI}[1] = b)\ \&\ !(\mathit{RMI}[3] = b)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[2] & = & c -> !(\mathit{RMI}[1] = c)\ \&\ !(\mathit{RMI}[3] = c)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[2] & = & d -> !(\mathit{RMI}[1] = d)\ \&\ !(\mathit{RMI}[3] = d)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[3] & = & a -> !(\mathit{RMI}[1] = a)\ \&\ !(\mathit{RMI}[2] = a)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[3] & = & b -> !(\mathit{RMI}[1] = b)\ \&\ !(\mathit{RMI}[2] = b)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[3] & = & c -> !(\mathit{RMI}[1] = c)\ \&\ !(\mathit{RMI}[2] = c)) \\
\mathit{SPEC}\ AG(\mathit{RMI}[3] & = & d -> !(\mathit{RMI}[1] = d)\ \&\ !(\mathit{RMI}[2] = d))
\end{array}
$$

(iv) The number of reservations (*res*) equals the number of rooms occupied by guests plus the number of reservations in progress (*underway*): the CTL formula (!(RMI[1] = n)) equals 1 if room 1 is occupied by some guest, otherwise it is 0. Thus the number of rooms occupied by guests is given by $(!(RMI[1] = n)) + (!(RMI[2] = n)) + (!(RMI[3] = n))$. The CTL formula describing the generalized integrity constraint, therefore, is as follows.

$$
\begin{array}{l}
\mathit{SPEC}\ AG(\mathit{res}\ =\ \mathit{underway}\ +\ (!(\mathit{RMI}[1] = n))\ + \\
\qquad\qquad (!(\mathit{RMI}[2] = n))\ +\ (!(\mathit{RMI}[3] = n)))
\end{array}
$$

(v) The set of rooms with status *Taken* equals the set of rooms assigned to guests and the rooms in the set *acquired*: this constraint is represented by the 3 CTL formulae given below. The first formula means (i) the status of room 1 is 1 (signifying that the room is *Taken*) implies that either room 1 is occupied by some guest or it is in the set *acquired*, and conversely, (ii) if room 1 is occupied by some guest or it is in the set *acquired*, its status is 1. The second and third formulae, which respectively apply for room 2 and room 3, can be similarly interpreted.

$SPEC\ AG(ST[1] <-> (!(RMI[1] = n)) \mid acquired[1])$
$SPEC\ AG(ST[2] <-> (!(RMI[2] = n)) \mid acquired[2])$
$SPEC\ AG(ST[3] <-> (!(RMI[3] = n)) \mid acquired[3])$

(vi) The rooms in the set *acquired* are not the ones assigned to guests. The 3 CTL formulae given below ensure this constraint. The first formula means that if room 1 is in the set acquired, then it is not assigned to some guest. The second and third formulae correspond to room 2 and 3 respectively.

$SPEC\ AG(acquired[1] -> \ (RMI[1] = n))$
$SPEC\ AG(acquired[2] -> \ (RMI[2] = n))$
$SPEC\ AG(acquired[3] -> \ (RMI[3] = n))$

Consistent Execution Property

The consistent execution property requires the original integrity constraints (specified in Section 7.1.1) to hold after the completion of all *Reserve* transactions. Note that the first three original integrity constraints are invariants in the decomposed model and they hold at every state (as shown by items (i), (ii) and (iii) in Section 7.3.3 under the subsection Generalized Integrity Constraints). The last two original integrity constraints are relaxed and the specification of these relaxed integrity constraints correspond to items (iv) and (v) in Section 7.3.3. Thus for the consistent execution property we need to show that when all the *Reserve* transactions complete, the original integrity constraints corresponding to only items (iv) and (v) are satisfied.

Before specifying the consistent execution property we need to show the specification of the original integrity constraints corresponding to items (iv) and (v). These are as follows.

(iv) The number of reservations equals the number of rooms occupied by the guests. The corresponding CTL specifications are given below:

$$SPEC\ AG(res = (!(RMI[1] = n)) + (!(RMI[2] = n)) + (!(RMI[3] = n)))$$

(v) The set of rooms with status taken equals the set of rooms assigned to guests. The CTL formula is:

$$SPEC\ AG(ST[1]\ <->\ !(RMI[1] = n))$$
$$SPEC\ AG(ST[2]\ <->\ !(RMI[2] = n))$$
$$SPEC\ AG(ST[3]\ <->\ !(RMI[3] = n))$$

When all the *Reserve* transactions complete, the following conditions hold: (a) for each step $R1$ either a step $R3$ or a compensating step $CompR1$ has executed. (b) for each step $R2$ either a step $R3$ or a compensating step $CompR2$ has executed. Thus at the completion of the *Reserve* transaction the following constraints hold corresponding to the conditions(a) and (b).

$$r1 = r3 + compr1$$
$$r2 = r3 + compr2$$

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

$$SPEC\ AG((r1 = r3 + compr1)\ ->$$
$$\quad res\ =\ (!(RMI[1] = n))\ +\ (!(RMI[2] = n))\ +\ (!(RMI[3] = n)))$$
$$SPEC\ AG((r1 = r3 + compr1)\ \&\ (r2 = r3 + compr2)\ ->$$
$$\quad (ST[1]\ <->\ !(RMI[1] = n))\ \&\ (ST[2]\ <->\ !(RMI[2] = n))\ \&\ (ST[3]\ <->\ !(RMI[3] = n)))$$

Sensitive Transaction Isolation Property

The sensitive transaction isolation property involves verifying that transaction *Report* is not executed when the set *acquired* is not empty. The CTL formula shown below specifies this property.

$$SPEC\ AG((acquired[1]\ |\ acquired[2]\ |\ acquired[3])\ ->\ (ReportJustExecuted\ =\ N))$$

Semantic Atomicity Property

The semantic atomicity property ensures that when a transaction has been partially executed, then it will eventually complete. Our model has the capability of executing a finite number of *Reserve* transactions. Semantic atomicity involves showing that in any state, if the preconditions of $R2$, $R3$, $CompR2$ or $CompR1$ are not satisfied, then in that state there are no outstanding incomplete *Reserve* transactions. The corresponding CTL formula is given below.

$SPEC\ AG(!((underway > 0)\ \&\ (r2 < maxR)\ \&$
$(((r\_inp = 1)\ \&\ !ST[1])\ \mid\ ((r\_inp = 2)\ \&\ !ST[2])\mid((r\_inp = 3)\ \&\ !ST[3])))\ \&$
$!((underway > 0)\ \&\ (r3 < maxR)\ \&$
$!(RMI[1] = g\_inp)\ \&\ !(RMI[2] = g\_inp)\ \&\ !(RMI[3] = g\_inp)\ \&$
$(((pr\_inp = 1)\ \&\ acquired[1])\ \mid\ ((pr\_inp = 2)\ \&\ acquired[2])\ \mid$
$((pr\_inp = 3)\ \&\ acquired[3])))\ \&$
$!(res > 0\ \&\ underway > 0\ \&\ (compr1 < maxR))\ \&$
$!((((pr\_inp = 1)\ \&\ acquired[1])\ \mid\ ((pr\_inp = 2)\ \&\ acquired[2])\mid$
$((pr\_inp = 3)\ \&\ acquired[3]))\ \&\ (compr2 < maxR)) -> (r1 = r3 + compr1)\ \&$
$(r2 = r3 + compr2))$

7.3.4 Analysis and Conclusions

We created models which allow at most 1, 2 or 3 *Reserve* transactions to execute. The SMV model checker was able to verify all the properties and generalized invariants for each of these models. The time taken to verify these models were 4 seconds, 7 seconds and 16 seconds on a SPARC 20. Unlike the theorem proving approach, this verification process was completely automated and did not require any help from the user.

The SMV model checker is relatively easy to use; its interface is well defined and the details of its working are hidden from the user. Based on our experience, our conclusion is that people with some programming skills will have no difficulty in using this model checker. The specification of the input accepted by the model checker is straightforward, and the outputs can be interpreted easily.

This preliminary study reveals that the SMV model checkers appears to be a good tool for verifying semantic-based decomposition. The biggest challenge of this task is in developing finite state abstractions of the application being verified. Unfortunately, the problem of developing finite state abstractions is application specific and we cannot give a generalized algorithm for this purpose. However, once an appropriate finite model has been developed, verification of the model is completely automated.

Chapter 8

Contributions and Future Research

8.1 Contributions

The traditional model which relies on the syntactic correctness criterion of serializability cannot be used for application domains having additional requirements like performance, autonomy, and security. In order to meet these additional requirements, transactions cannot be executed atomically but need to be decomposed into steps. Decomposition of a transaction into steps results in the loss of atomicity, consistency and isolation properties. Once these properties are violated, we can no longer give assurance of the correct behavior of the application using syntactic-based correctness approaches. In this thesis we propose a new approach for transaction decomposition which uses the semantics of transactions to reason about correct behavior and is suitable for the above mentioned application domains.

The decomposition of a transaction into steps results in the loss of properties which form the foundation for the traditional transaction processing model. Our first contribution aims at remedying this loss by proposing a set of replacement properties, namely, semantic atomicity, consistent execution and sensitive transaction isolation. These properties, which form the basis of semantic correctness, ensure that all partially executed transactions complete, database consistency is preserved after all transactions complete, and no transaction displays any inconsistent data to the user. In addition to the semantic correctness properties we propose a fourth property, the composition property which ensures that a decomposition correctly models the original transaction.

The semantic correctness properties are defined in terms of semantic histories in which the steps of transaction execute serially. However, serial execution of steps results in poor performance. Our second contribution is proposing notions of correct concurrent executions and developing concurrency control algorithms for long duration, multidatabase and multilevel secure domains. These algorithms not only produce correct concurrent executions but also satisfy the domain specific requirements. In the long duration case, our model provides greater concurrency and allows a larger class of histories than the traditional model. Most of the other models proposed in the long duration domain [AAS93, FÖ89, GMS87, KS88, KS94] assume that the application developer is responsible for coming up with a correct decomposition and does not provide any guidance of how to perform this decomposition or reason about its correctness. In the multidatabase case, our model achieves greater autonomy, provides greater concurrency and allows a larger class of histories than the traditional model. The other models proposed in this domain are not satisfactory because some require a centralized global transaction manager [BS88, MRB+92, MRKS92], some make assumptions about the nature of integrity constraints [DE89, GM91] and others have high execution time overhead [BS88, MRB+92, RKS95]. In the multilevel secure case, our approach allows a larger class of multilevel transactions to be processed, requires lesser synchronization between security levels and achieves more concurrency than the approaches based on the traditional transaction processing model.

Our third contribution is in demonstrating how example applications can be analyzed to show the existence of the semantic correctness properties. The semantic correctness properties generate a set of proof obligations that must be successfully discharged. Convincing formal arguments must be made to discharge these proof obligations; this motivated us to investigate the use of a suitable specification language for capturing the semantics of the applications and specifying the semantic correctness properties as the use of a specification language allows for the standard formal analysis techniques of verification and refinement. Although not adequate, the Object Z specification language was the closest to fulfilling our needs. Object Z allowed us to specify transactions, steps, integrity constraints, relaxed integrity constraints, the sensitive transaction isolation property, the consistent execution property, and the composition property. Using example applications we show how typical applications from each application domain can be analyzed for satisfaction of the properties in Object Z.

Since the Object Z specification language does not have adequate tool-support, all the analysis using Object Z was performed manually. However, hand analysis may not be feasible for large complex applications. Thus our fourth contribution involves investigating the applicability of existing mechanized software verification tools for automated analysis of applications. The two common approaches of automated software verification are theorem proving and model checking. Specifically, we looked at the B-Toolkit for the theorem proving approach and the SMV for the model checking approach. The B-Toolkit allows the specification and verification of only those properties which are invariant. Using the B-Toolkit we could only check whether the decomposed specification satisfies the relaxed integrity constraints. This verification, however, was not completely automated as it required aid from the user in the form of user theories. Moreover, since the B-Toolkit does not support temporal features, we could not specify or verify the semantic correctness properties. The SMV model checker, on the other hand, does not have this shortcoming; the properties verified by the SMV must be specified in Computational Tree Logic (CTL) which is a branching time temporal logic. However, model checkers can verify only finite state machines;

software, in general, are infinite state machines and cannot be directly verified by model checkers. This required us to create finite state abstractions of the decomposed application which could be verified by the model checker. On the finite state abstraction of the example we studied we could verify all of the semantic correctness properties.

8.2 Future Research

8.2.1 Implementation

In this thesis we have proposed concurrency control algorithms for the different types of application domains that we have studied. From our study it appears that the semantic-based approach has enormous potential. However, without an implementation it is not possible to evaluate completely the benefits and drawbacks of this approach. Towards this end, we plan to study existing implementations of semantic-based decomposition and develop suitable implementations for each application domain.

8.2.2 Performance Evaluation

One of the benefits of decomposing transactions into steps is an increase in performance. For each application domain, we plan to develop metrics and studies to evaluate the extent to which a semantic decomposition is successful in achieving this goal. One measure compares the allowed interleaving of steps in a decomposition to all possible interleavings, and thus describes the restrictiveness of a particular decomposition. A second measure compares the performance of a decomposition to that of a standard implementation in which transactions are subject only to the syntactic requirement of serializability. The second measure is clearly more important, in that it captures the benefit of our semantic transaction processing model over the standard approaches used in commercial databases today.

Once the implementations have been developed, the performance of semantic-based approach can also be evaluated empirically; this we plan to do by measuring its execution time on some standard benchmark programs and comparing the results with those of existing implementations.

8.2.3 Recovery

An important property of transactions is durability – once a transaction commits all its changes are made permanent. However a system may have one or more malicious transactions; other good transactions may have read from these committed malicious transactions. Once these malicious transactions are discovered their effects on the database and on other transactions must be removed. A syntactic-based approach might require undoing the malicious transaction and all the transactions that have executed after the malicious transaction and then redoing all the good transaction. This may not be acceptable from the performance point of view. A semantic-based approach looks promising in that it may reduce the number of transactions that have to be undone and redone and at the same time ensure that the database is eventually restored to an acceptable consistent state. However this needs to be investigated further.

8.2.4 Survivability

Prevention of attacks on a database system is not enough for ensuring security; attacks will occur and and it is necessary to live through and recover from successful attacks [AJMB97]. Consider the simple example of an attacker corrupting some data in the database. When such an attack is detected, it is desirable that the system continues to function while the damage is being repaired. In order to minimize the loss in the system's functioning, the degree of damage caused by a corrupt data and the degree of repair of a corrupt data must be assessed. This is because damage to some data may not have any serious consequences, and it may be possible to tolerate this damage. Similarly, even though the exact value of a corrupt data is not restored, it may be possible for the system to operate with some approximate value. These approximate values may not satisfy the database integrity constraints and a new relaxed set of integrity constraints must be defined. The system continues to function (maybe to a reduced extent) when these relaxed integrity constraints are satisfied. Eventually repair transactions, which are responsible for repairing damaged data items, must be executed to restore consistency. Semantics of the application can be used to address this problem of survivability.

8.2.5 Extension to Active Database Applications

One area that researchers [SKD95] anticipate will benefit from formal method techniques and tools is active databases. Active databases are expected to facilitate the design and maintenance of business rules, ensure the enforcement of business rules in applications, and enhance their performance. However, application developers are reluctant to use active database because of the uncertainty associated with how a set of individual triggers acting on their own react with each other and with other application programs. This uncertainty stems mainly from the absence of a formal design methodology that guides the application developers in the design of active database applications. The use of formal method in designing triggers and transactions will give a basis for analyzing the interactions of triggers and transactions in an active database. This will also enable the application developer to decompose the transactions and/or relax the integrity constraints to improve performance. We also plan to look into the security aspects of active database applications.

8.2.6 Extension to Workflow Applications

One more area which we plan to investigate is the applicability of formal methods in workflow systems. Workflow systems are gaining attention owing to their perceived ability to integrate heterogeneous distributed applications. However, there is a significant lack of tools for designing, implementing and maintaining such systems and formal methods show promise in this area.

Bibliography

Bibliography

[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.

[Abr96] J. R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[AG93] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[AJMB97] P. Ammann, S. Jajodia, C. D. McCollum, and B. T. Blaustein. Surviving information warfare attacks on databases. In *Proceedings of the Symposium on Security and Privacy*, pages 164–174, Oakland, CA, May 1997.

[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 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.

[AK95] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via non-masking fault-tolerance. In *Proceedings of Symposium on Reliable Distributed Systems*, pages 174–185, Bad Neuenahr, Germany, September 1995.

[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.

[BR92] B.R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

[BS88] Y. Breitbart and A. Silberschatz. Multidatabase update issues. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 135–142, June 1988.

[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.

[CR94] P. K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[DD90] D. Duke and R. Duke. Towards a semantics for Object Z. In D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z*, volume 428 of *Lecture Notes in Computer Science*, pages 242–262. Springer-Verlag, 1990.

[DE89] W. Du and A.K. Elmagarmid. Quasi serializability: A correctness criterion for global concurrency control in interbase. In *Proceedings of the International Conference on Very Large Databases*, pages 347–355, Amsterdam, Netherlands, 1989.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[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.

[GM91] H. Garcia-Molina. Global consistency constraints considered harmful for heterogeneous database systems. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 248–250, Kobe, Japan, April 1991.

[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.

[Her87] M. Herlihy. Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers*, 36(4):443–448, April 1987.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12(10):576–580, October 1969.

[HW91] M. P. Herlihy and W. E. Weihl. Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991.

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

[JM87] S. Jajodia and C. Meadows. Managing a replicated file in an unreliable network. In *Proceedings of 3rd IEEE International Conference on Data Engineering*, pages 396–404, Los Angeles, CA, February 1987.

[KLS90] H.F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, 1990.

[KS88] H. F. Korth and G. D. Speegle. Formal model of correctness without serializability. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 379–386, June 1988.

[KS94] H. F. Korth and G. Speegle. Formal aspects of concurrency control in long-ouration transaction systems using the NT/PV model. *ACM Transactions on Database Systems*, 19(3):492–535, September 1994.

[LBS96] K. Lano, J. Bicarregui, and A. Sanchez. Using B to design and verify controllers for chemical processing. In H. Habrias, editor, *Proceedings of the 1st Conference on the B Method*, Nantes, November 1996.

[LKS91a] E. Levy, H. F. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 88–97, Denver, CO, May 1991.

[LKS91b] E. Levy, H. F. Korth, and A. Silberschatz. A theory of relaxed atomicity. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 95–109, August 1991.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, CA, 1994.

[Lyn83] N. A. Lynch. Multilevel atomicity—A new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[MCFP96] C. Morpain, M. Cart, J. Ferrie, and J. F. Pons. Maintaining database consistency in presence of value dependencies in multidatabase systems. In *ACM SIGMOD Record*, pages 459–468, Montreal, Canada, June 1996.

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

[MRB+92] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. The concurrency control problem in multidatabases: Characteristics and solutions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 288–297, San Diego, CA, 1992.

[MRKS91] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Non-serializable executions in heterogeneous distributed database systems. In *Proceedings of the International Conference on Parallel and Distributed Systems*, pages 245–252, Miami Beach, FL, December 1991.

[MRKS92] S. Mehrotra, R. Rastogi, H. Korth, and A. Silberschatz. A transaction model for multidatabase systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 56–63, 1992.

[NZ96] M. H. Nodine and S. B. Zdonik. The impact of transaction management on object-oriented mulidatabase views. In O. A. Bukhres and A. K. Elmagarmid, editors, *Object-Oriented Multidatabase Systems*, chapter 3, pages 57–104. Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[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.

[PST91] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice-Hall, New York, 1991.

[RAJ] I. Ray, P. Ammann, and S. Jajodia. Using semantic correctness in multidatabases to achieve local autonomy, distribute coordination and maintain global integrity. Submitted for Publication.

[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.

[SH94] A. C. Storey and H. Haughton. A strategy for the production of verifiable code using the B method. In *FME '94 : Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 346–365. Springer-Verlag, 1994.

[SKD95] E. Simon and A. Kotz-Dittrich. Promises and realities of active database systems. In *Proceedings of the International Conference on Very Large Databases*, pages 642–653, Zurich, Switzerland, September 1995.

[SLJ88] L. Sha, J. P. Lehoczky, and E.D. Jensen. Modular concurrency control and failure recovery. *IEEE Transactions on Computers*, 37(2):146–159, February 1988.

[Sor94] I. H. Sorenson. Demonstration of the B-Toolkit. In *Proceedings of the 6th Nordic Workshop on Programming Theory*, 1994.

[SSV92] D. Shasha, E. Simon, and P. Valduriez. Simple rational guidance for chopping up transactions. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 298–307, San Diego, CA, June 1992.

[Wei84] W. E. Weihl. *Specification and Implementation of Atomic Data Types.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1984.

[Wei88] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[Wor96] J. Wordsworth. *Software Engineering with B.* Addison-Wesley, September 1996.

[WR92] H. Wachter and A. Reuter. The ConTract model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kauffman, San Mateo, CA, 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 15213, April 1996. To appear in Science of Computer Programming.

Appendices

Appendix A

Example Decomposition of Long Duration Transactions

As a case study we look at a hotel database example which has a long duration transaction. In this work we use the Object Z specification language [DD90] to formally specify the application. Object Z extends the Z specification language [PST91] to include object-oriented features and has a trace-based semantics. Object Z , like Z, is a model-oriented specification language and it is based on set-theory and predicate calculus. Table A.1 lists the Object Z notations used in this example and in the examples presented in Appendices B and C.

Table A.1: Relevant Object Z Notations

$\mathbb{Z}$	Set of Integers
$\mathbb{N}$	Set of Natural Numbers
$\mathbb{P}\,A$	Powerset of Set A
$\varnothing$	Empty Set
$\downarrow A$	Set of all possible objects from A and any derived class
$x \mapsto y$	Ordered Pair $(x,\ y)$
$A \nrightarrow B$	Partial Function from A to 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
Δx	Before and After State of State Variable x
op	Identifier denoting the operation
$\Box P$	P always holds
$\Diamond P$	Eventually P holds
$\bigcirc P$	In the next state P holds

We use the *class* definition of Object Z to specify the hotel database. Syntactically, a class definition is a named box, in which the constituents of the class are defined and related. The constituents of a class include type and constant definitions, state schema, initialization schema, operation schemas, and history invariants. A schema is a two-dimensional notation used in Object Z to specify the state as well as operations on the state. The state schema is nameless and consists of two parts: the top part contains declarations of the state variables, and the bottom part specifies the constraints on these variables. The initialization schema, named $INIT$, defines the possible initial states. An operation schema, which is named after the operation, also consists of two parts: the bottom part specifies the operation using preconditions and postconditions and the top part contains declarations of the variables used in the bottom part. A history invariant is a predicate defined over a sequence of operations which further constrains the behavior of the objects. An Object Z specification appears in figure A.1. The specification assumes two basic types, *Guest* and *Room*, which enumerate all possible guests and all possible hotel rooms, respectively. The enumerated type *Status* lists the two values, *Available* and *Taken*, which indicates the status of each room.

The class *Hotel* models the hotel database. The database objects may be constants or variables. The hotel database has a constant *total* which is the number of rooms in the hotel. The hotel database has three variables, namely, *res*, RM and ST which are declared in the state schema. The natural number *res* counts current reservations, and the partial injection RM relates guests to rooms. Our particular example does not allow guests to register multiple times, which is reflected in the fact that RM is an injective function. The example could be modified easily with different constraints. The partial function ST records the status of each room. Additional integrity constraints on the objects in hotel database appear in the bottom part of the state schema. There are two such constraints:

1. $\#RM = res$. The number of guests who have been assigned rooms (the size of

the RM function) equals the total number of reservations (res).

2. $\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM$. The set of rooms that are taken ($\mathrm{dom}(ST \rhd \{Taken\})$) is exactly the set of rooms reserved by guests ($\mathrm{ran}\, RM$).

The three operation schemas *Reserve*, *Cancel* and *Report* describe the three transactions in the hotel database. *Reserve* reserves a room for guest $g?$ and produces as output a room assignment $r!$. *Reserve* has a precondition that there must be fewer than *total* reserved rooms and $g?$ must be a new guest. Since the domain of RM is the set of guests with reservations, the latter is captured by checking that $g? \notin \mathrm{dom}\, RM$. *Reserve* has a postcondition that some room $r!$ with status *Available* is chosen, the number of reservations is incremented, the status of $r!$ is changed to *Taken*, and the ordered pair $g? \mapsto r!$ is added to the function *RM*. *Cancel* cancels a reservation for guest $g?$. *Cancel* has a precondition that $g?$ is a guest and a postcondition that res is decremented, $g?$ is removed from the domain of the function RM, and the status of the room for $g?$ is changed to *Available*. *Report* has no precondition, and merely produces the state components ST and RM as outputs.

Since the role of initialization is peripheral to our analysis, we omit initialization schemas here. Instead, we assume that the database has been initialized to a consistent state. As no history invariants are needed to restrict the execution of operations, we do not specify any history invariant.

A.1 Decomposition of Long Duration Transaction in the Hotel Example

We now provide a valid decomposition of the hotel database which satisfies all the necessary properties described in Chapter 4. The class *HotelD* (short for *Hotel Decomposition*) in figure A.2 specifies this valid decomposition.

We make the integrity constraints more general by adding the auxiliary variables *underway* and *acquired*. *underway* is a natural number which denotes the reservations that have been partially processed. The auxiliary variable *acquired* denotes the set of rooms that have been taken but which have not yet been assigned to guests. The declarations of these auxiliary variables appear in the top part of the state schema. The two generalized integrity constraints appear at the bottom of the state schema. They are: (i) $\#RM + underway = res$ and (ii) $\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM \cup acquired$. The third constraint, appearing at the bottom of the state schema, signifies that the rooms in the set *acquired* are not the ones assigned to guests.

Since the steps and compensating steps now comprise the operations of the decomposed hotel database, we specify operation schemas corresponding to each type of step and compensating step. Only the *Reserve* transaction is decomposed into three steps, namely $R1$, $R2$ and $R3$. $R1$ has a precondition that that there must be fewer than *total* rooms. The postcondition of $R1$ increments *res* and *underway*. The precondition of $R2$ is that the room to be assigned to the guest $r!$ is *Available*. The postcondition of $R2$ changes the status of $r!$ to *Taken* and inserts $r!$ in the set *acquired*. The precondition of $R3$ is that $g?$ must be a new guest. The postcondition of $R3$ inserts the ordered pair $g? \mapsto r!$ to the function RM, removes $r!$ from the set *acquired* and decrements *underway*. The compensating step $CompR1$ semantically undoes the effect of $R1$. The preconditions of $CompR1$ are that the variables *res* and *underway* must be positive. The postcondition of $CompR1$ decrements *res* and *underway*. The compensating step $CompR2$ semantically undoes the effects of $R2$. The precondition of $CompR2$ is that $r!$ must be in the set *acquired*. The postconditions of $CompR2$ remove $r!$ from the set *acquired* and change the status of $r!$ to *Available*. *CancelD* and *ReportD* represent the single steps of the *Cancel* and *Report* transaction respectively; the specifications of these steps are identical to the corresponding transactions.

A.2 Properties of the Decomposition

The Composition Property

Proof Obligation:

$$Hotel \wedge ((R1 \fatsemi R2 \fatsemi R3) \setminus (underway, acquired, underway', acquired')) \Leftrightarrow Reserve$$

Proof:

The left hand side of the equivalence is obtained by schema composition [PST91] followed by schema conjunction and evaluates to:

$$
\begin{array}{|l}
\multicolumn{1}{l}{HotelState \wedge ((R1 \fatsemi R2 \fatsemi R3) \setminus (underway, acquired, underway', acquired'))} \\
\hline
\Delta(res, ST, RM) \\
g? : Guest \\
r! : Room \\
\hline
res < total \\
g? \notin \operatorname{dom} RM \\
ST(r!) = Available \\
res' = res + 1 \\
ST' = ST \oplus \{r! \mapsto Taken\} \\
RM' = RM \cup \{g? \mapsto r!\} \\
\#RM + underway = res \\
\#RM' + underway' = res' \\
\operatorname{dom}(ST \rhd \{Taken\}) = \operatorname{ran} RM \cup acquired \\
\operatorname{dom}(ST' \rhd \{Taken\}) = \operatorname{ran} RM' \cup acquired' \\
\operatorname{ran} RM \cap acquired = \varnothing \\
\operatorname{ran} RM' \cap acquired' = \varnothing' \\
\hline
\end{array}
$$

The schema *HotelState* represents the state schema (nameless schema) of class *Hotel* shown in figure A.1. The predicate part of this schema is equivalent to the predicate part of the *Reserve* schema given in Figure A.1; hence the two schemas are equivalent.

The Sensitive Transaction Isolation Property

Proof Obligation:

The sensitive transaction *Report* does not output any inconsistent data.

Proof:

The Sensitive Transaction Isolation Property is achieved by construction. Consider *Report* transactions. We compute the subset of the original invariants that must hold as a precondition for *Report*. We obtain these from *Hotel* by hiding the state variables *not* involved in producing the outputs of *Report*. The only state variable not involved in producing the output of *Report* is *res*. Hiding the variable *res* from *Hotel* produces the following schema:

$$
\begin{array}{l}
\textit{HotelState} \setminus (\textit{res}) \\
\hline
RM : \textit{Guest} \rightarrowtail\!\!\!\!\!\!+ \; \textit{Room} \\
ST : \textit{Room} \nrightarrow \textit{Status} \\
\hline
\exists\, \textit{res} : \mathbb{N} \bullet (\\
\#RM = \textit{res} \wedge \\
\mathrm{dom}(ST \rhd \{\textit{Taken}\}) = \mathrm{ran}\, RM)
\end{array}
$$

The constraint simplifies to

$$\mathrm{dom}(ST \rhd \{\textit{Taken}\}) = \mathrm{ran}\, RM$$

The constraint is implied by a generalized integrity constraint in $\hat{I}$, namely,

$$\mathrm{dom}(ST \rhd \{\textit{Taken}\}) = \mathrm{ran}\, RM \cup \textit{acquired}$$

and $\textit{acquired} = \varnothing$. Hence we include $\textit{acquired} = \varnothing$ as a precondition for *ReportD*.

For the hotel example, we do not have any multistep transaction 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.

The Consistent Execution Property

Proof Obligation:

For any correct complete semantic history H, if the initial state satisfies the integrity constraints listed in the state schema of *Hotel*, the final state also satisfies those constraints.

Proof:

For the hotel database system, the original integrity constraints in I are satisfied when $acquired = \varnothing$ and $underway = 0$. Thus we have to prove that if the initial state of a correct complete semantic history satisfies $acquired = \varnothing$ and $underway = 0$, the final state of the history will also satisfy $acquired = \varnothing$ and $underway = 0$.

Let $r1$, $r2$, $r3$, $compr1$, $compr2$ be the number of steps of type $R1$, $R2$, $R3$, $CompR1$ and $CompR2$ respectively present in any complete history. The variable *underway* is incremented by steps of type $R1$ and it is decremented by steps of type $R3$ and $CompR1$. Thus *underway* is given by the following expression

$$underway = r1 - (r3 + compr1) \quad \cdots \quad (1)$$

The variable *acquired* is modified by steps of type $R2$, $R3$ and $CompR2$. In step of type $R2$ a room is added to the set *acquired* and in steps of type $R3$ or compensating steps of type $CompR2$ the room is taken out from the set *acquired.* Thus we can write,

$$\mid acquired \mid = r2 - (r3 + compr2) \quad \cdots \quad (2)$$

In a complete history all the reserve transactions have completed execution. Therefore corresponding to each step of type $R1$ in the history there is a step of type $R3$ or a compensating step of type $CompR1$. Thus

$$r1 = r3 + compr1 \quad \cdots \quad (3)$$

Similarly corresponding to each $R2$ there is either a step of type $R3$ or a compensating step of type $CompR2$. Thus

$$r2 = r3 + compr2 \quad \cdots \quad (4)$$

From (1),(2),(3) and (4) we can conclude that in a complete history, 1) $underway = 0$ and 2) $\mid acquired \mid = 0$ or $acquired = \varnothing$. Hence the hotel database has the consistent execution property.

The Semantic Atomicity Property

Proof Obligation:

Any correct partial semantic history H_p consisting of incomplete *Reserve* transactions is a prefix of some correct complete semantic history defined over the same set of transactions.

Proof:

The proof that the hotel database has the semantic atomicity property proceeds by induction. Suppose H_p consists of n incomplete *Reserve* transactions. We show how an incomplete reserve transaction can be compensated, thereby decreasing the number of incomplete *Reserve* transactions from n to $n-1$. By repeated applications of our argument, it is possible to reduce the number of incomplete *Reserve* transactions to zero, at which point the history is a correct complete semantic history H_c. The history H_p is a prefix of the correct complete semantic history H_p.

In the following we show how an incomplete *Reserve* transaction can be completed. Suppose the *Reserve* transaction has completed only step $R1$. In this case we need to prove that it will be possible to execute the corresponding $CompR1$, no matter what steps of other transactions execute between $R1$ and $CompR1$, and complete the *Reserve* transaction. The step $CompR1$ can be executed if its preconditions are satisfied. The step $CompR1$ has two preconditions, (i) $res > 0$ and (ii) $underway > 0$. The variable res is incremented by steps of type $R1$ and it is decremented by steps of type $CompR1$ and $CancelD$. Steps of type $CancelD$ cancels a previously completed reservation. Thus the *Cancel* transaction decrements the value of res which was incremented by some other previously completed *Reserve* transaction. Steps of type $CompR1$ belonging to other transactions also decrement the value of res; however this decrement is done to neutralize the increment caused by step $R1$ of the corresponding reserve transaction. Thus the precondition of $CompR1$ will always be satisfied if $CompR1$ is executed after step $R1$ of the same *Reserve* transaction.

The variable *underway* is incremented by steps of type $R1$ and is decremented by steps of type $R3$ and $CompR1$. The steps of other transactions of type $R3$ or $CompR1$ decrements the variable *underway* which was incremented by the corresponding $R1$. Hence even if steps of other transactions appear after $R1$, the precondition $underway > 0$ of $CompR1$ will always be satisfied.

Similarly it can be proved that if the *Reserve* transaction has completed the first two steps of type $R1$ and $R2$, it will be possible to execute compensating steps of type $CompR2$ and $CompR1$ and complete the *Reserve* transaction.

The Valid Successor Set Property

A possible successor set description for the hotel example is given in Table A.2. We show that this description is valid by proving the valid successor set property.

Proof Obligations:

(a) All correct successor set histories generated from the successor set description in Table A.2 satisfy the semantic atomicity property.

(b) Any complete correct successor set history generated from Table A.2 is

Table A.2: Successor Sets for the Hotel Example

Successor Set of Type of Step	Types of Steps in Successor Set
$SS(R1)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR1$, $CompR2$
$SS(R2)$	$R1$, $R2$, $R3$, $CancelD$, $CompR1$, $CompR2$
$SS(R3)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR1$, $CompR2$
$SS(ReportD)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR1$, $CompR2$
$SS(CancelD)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR1$,$CompR2$
$SS(CompR1)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR1$,$CompR2$
$SS(CompR2)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR1$,$CompR2$

equivalent to a correct complete semantic history generated from the generalized specifications.

Proof of (a):

In the first part of the valid successor set property, it is required to show that the specification generated from successor set description has the semantic atomicity property. Note that $CompR1$ and $CompR2$ are in the successor set of all types of steps. This means that the successor set imposes no constraint on the execution of the compensating steps. As before we can show that the preconditions of these steps will always be satisfied no matter what other transactions execute in between the steps of the *Reserve* transaction and the corresponding compensating steps and so any partially executed *Reserve* transaction can always complete.

Proof of (b):

In the second part we must show that a correct complete successor set history is equivalent to a correct complete history generated from the specification with generalized invariants. Consider any correct complete successor set history H_s. From H_s we construct a semantic history H_g (corresponding to the specification with generalized invariants) as follows: (i) make the initial state of H_g the same as H_s, (ii) for any step in H_s, include the same step in H_g, (iii) add the precondition *acquired* $= \varnothing$ to any and all occurrence(s) of steps of type *ReportD* in H_1.

Now we must show that H_g is a correct semantic history. Let us consider the precondition ($acquired = \varnothing$) that was added to *ReportD* in H_g. From the specifications of $R1$, $R2$, $R3$, *CancelD*, and *ReportD* we see that the only situation in which the precondition of a step of type *ReportD* is not satisfied, is when at least one step of type $R2$ has been executed but the next step, which is of type $R3$ or $CompR2$, has not yet been executed. However, as $ReportD \notin SS(R2)$, a step of type *ReportD* will never be allowed to execute between a step of type $R2$ and a step of type $R3$ or $CompR2$. Thus in H_1 $acquired = \varnothing$ is always satisfied before *ReportD* is executed.

Note that the set of operations that modify the database state are the same for steps in H_s and H_g. The initial states are the same for both the histories. So the state obtained by applying a step in H_s is the same as that obtained by applying H_g. Thus any correct successor set history is equivalent to a correct semantic history generated from the specification containing the generalized integrity constraints.

$[Guest, Room]$

$Status ::= Available \mid Taken$

Hotel

$total : \mathbb{N}$

$res : \mathbb{N}$
$ST : Room \nrightarrow Status$
$RM : Guest \rightarrowtail\!\!\!\!\!\nrightarrow Room$

$\#RM = res$
$\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM$

Reserve

$\Delta(res, ST, RM);\ g? : Guest;\ r? : Room$

$res < total;\ g? \notin \mathrm{dom}\, RM$
$ST(r?) = Available$
$res' = res + 1$
$ST' = ST \oplus \{r? \mapsto Taken\}$
$RM' = RM \cup \{g? \mapsto r?\}$

Cancel

$\Delta(res, RM, ST)$
$g? : Guest$

$g? \in \mathrm{dom}\, RM$
$res' = res - 1$
$RM' = \{g?\} \lhd RM$
$ST' = ST \oplus \{RM(g?) \mapsto Available\}$

Report

$currentST! : Room \nrightarrow Status$
$currentRM! : Guest \rightarrowtail\!\!\!\!\!\nrightarrow Room$

$currentST! = ST$
$currentRM! = RM$

Figure A.1: Initial Specification of the Hotel Database

HotelD

$res, underway : \mathbb{N}$
$ST : Room \nrightarrow Status$
$RM : Guest \rightarrowtail Room$
$acquired : \mathbb{P}\, Room$

$\#RM + underway = res$
$\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM \cup acquired$
$\mathrm{ran}\, RM \cap acquired = \varnothing$

R1

$\Delta(res, underway)$

$res < total$
$res' = res + 1$
$underway' = underway + 1$

R2

$\Delta(ST, acquired);\ r? : Room$

$underway > 0$
$ST(r?) = Available$
$ST' = ST \oplus \{r? \mapsto Taken\}$
$acquired' = acquired \cup \{r!\}$

R3

$\Delta(RM, acquired, underway)$
$g? : Guest;\ r? : Room$

$underway > 0$
$r? \in acquired$
$g? \notin \mathrm{dom}\, RM$
$RM' = RM \cup \{g? \mapsto r?\}$
$acquired' = acquired \setminus \{r?\}$
$underway' = underway - 1$

CompR1

$\Delta(res, underway)$

$res > 0;\ underway > 0$
$res' = res - 1$
$underway' = underway - 1$

CompR2

$\Delta(acquired, ST)$
$r? : Room$

$r? \in acquired$
$acquired' = acquired \setminus \{r!\}$
$ST' = ST \oplus \{r? \mapsto Available\}$

CancelD

$\Delta(ST, RM, res);\ g? : Guest$

$g? \in \mathrm{dom}\, RM$
$res' = res - 1$
$ST' = ST \oplus \{RM(g?) \mapsto Available\}$
$RM' = \{g?\} \mathbin{\lhd\!\!\!-} RM$

ReportD

$currentST! : Room \nrightarrow Status$
$currentRM! : Guest \rightarrowtail Room$

$acquired = \varnothing$
$currentST! = ST$
$currentRM! = RM$

Figure A.2: A Valid Decomposition for the Hotel Database

Appendix B

Example Decomposition of Global Transactions

We consider a simple banking example adapted from Mehrotra's work [MRKS92]. The banking enterprise consists of a set of branches and a head office. The head office, which is also a branch, performs additional functions. At each branch the local database has an object known as *accounts* which stores the set of accounts in that branch. Each account in the branch is associated with a balance, denoted *accbal*. The head office stores an additional database object, known as *banktotal*, which stores the total money in the bank. The banking enterprise has two local integrity constraints: the balance of each account is positive, and an account is associated with only one balance. The banking enterprise has one global integrity constraint that *banktotal* must equal the sum of money in all the branches of the bank.

The banking enterprise has a number of local and global transactions. The local transactions are *LDeposit*, *LWithdraw*, *LTransfer*, and *LAudit*. There are four types of global transactions, *GDeposit*, *GWithdraw*, *GTransfer*, and *GAudit*. The *LDeposit* transaction performs a deposit at an account in the head office. The precondition of *LDeposit* is that the account must exist in the head office. The postcondition of *LDeposit* increments the corresponding account balance and the object *banktotal*. *LWithdraw* transaction is defined similarly. *LTransfer* transaction transfers money from one account in a branch (or head office) to another account in the same branch (or head office). *LAudit* has no preconditions and produces as output the account balances of all the accounts in the branch. To perform a deposit in an account in a branch that is not the head office, the global transaction *GDeposit* must be executed. The *GDeposit* transaction increments the account balance and also increments the non local object *banktotal*. The global transaction *GTransfer* transfers money from an account in one branch to an account in a different branch. The global transaction *GAudit* prints the balances of accounts in all the branches.

In this example we consider an instance of the banking enterprise with one branch (Branch B) and one head office (Head Office H). The Object Z specification of the original banking application before the decomposition is given in figures B.1 and B.2. The class *Branch* describes the objects, local integrity constraints and the local transactions, global subtransactions associated with the branch. Global subtransactions perform a part of the work of the transaction and are not executed atomically. The class *HeadOffice* describes the objects, local integrity constraints local transactions and global subtransactions associated with the head office. Since the class *HeadOffice* is inherited from the class *Branch*, all local transactions and subtransactions, except $GWithdraw2$, $GDeposit2$ and $GAudit2$, specified in class *Branch* can be executed by the head office. The figure B.2 specifies the Banking Enterprise which in our case consists of one branch and one head office. The nameless schema of class *BankingEnterprise* lists the original integrity constraint. In addition, the class *BankingEnterprise* lists all possible transactions. Note that the global transactions are specified as conjunctions of two local subtransactions indicating that these two local subtransactions must be executed as one atomic action. Following the Object Z notation we use $B.accbal$, $H.accbal$ to refer to the object *accbal* in B and H respectively. Similarly, we use the notation $B.LTransfer$, $H.LTransfer$ to indicate the *LTransfer* operation associated with Branch B and Head Office H, respectively.

B.1 Decomposition of Global Transactions in the Bank Example

Instead of executing the global transactions as one atomic action, we decompose them into atomic steps such that each step executes at one site only. For example, the *GTransfer* transaction can be decomposed into two atomic steps $GTransfer1D$ and $GTransfer2D$. Similarly, for all other global transactions.

Once the transactions are decomposed into atomic steps, we need to specify compensating steps that will semantically undo the partially executed transactions. Thus we specify $GTrans1CompD$ which will semantically undo the actions of $GTransfer1D$. Similarly, we specify compensating steps $GWith1CompD$, $GDep1CompD$, which respectively compensate for $GWithdraw1D$ and $GDeposit1D$ respectively. The step $GAudit1D$ generates an output and so it is uncompensatable.

Note that the decomposition of global transactions into steps, violates the global integrity constraint that *banktotal* equals the sum of account balances in all the branches. To alleviate this problem we generalize the global integrity constraints using auxiliary variables. We associate an auxiliary variable with each type of step of a global transaction. These auxiliary variables are natural numbers which are incremented each time the corresponding step executes. For example, $tempDep1$, $tempDep2$, $tempDep1Comp$ are the auxiliary variables associated with $GDeposit1D$, $GDeposit2D$ and $GDep1CompD$ type of step respectively. Each time $GDeposit1D$ executes, it increments the variable $tempDep1$ by the amount of money to be deposited. Similarly, the variables $tempDep2$, $tempDep1Comp$ are incremented by $GDeposit2D$, $GDep1CompD$ respectively.

The generalized integrity constraint for the banking enterprise consisting of the Branch B and Head Office H is given in the bottom part of the state schema in figure B.5. Figures B.3 and B.4 specify the steps, local transactions executed at the branch and head office.

B.2 Properties of the Decomposition

Composition Property

Proof Obligations :

(i) $BEState \wedge ((H.GWithdraw1D \fatsemi B.GWithdraw2D) \setminus (aux, aux')) \Leftrightarrow GWithdraw$

(ii) $BEState \wedge ((H.GDeposit1D \fatsemi B.GDeposit2D) \setminus (aux, aux')) \Leftrightarrow GDeposit$

(iii) $BEState \wedge ((H.GAudit1D \fatsemi B.GAudit2D) \setminus (aux, aux')) \Leftrightarrow GAudit$

(iv) $BEState \wedge SelectTwoBanks \bullet ((c1?.GTransfer1D \fatsemi c2?.GTransfer2D) \setminus (aux, aux'))$
$\Leftrightarrow GTransfer$

where $BEState$ denotes the state schema of class $BankingEnterprise$ shown in figure B.2 and $aux = H.tempWith1, B.tempWith2, H.Audit1, B.Audit2, H.tempXfer1,$ $H.tempXfer2, B.tempXfer1, B.tempXfer2, H.tempDep1, B.tempDep2$ and aux' represents the corresponding primed components.

Proof of (i) :

The RHS of expression (i) evaluates to the following:

GWithdraw

$amt : \mathbb{N};\ acc : ACCOUNT$

$H.banktotal > amt?$
$acc? \in B.accounts$
$amt? < B.accbal(acc?)$
$H.banktotal' = H.banktotal - amt?$
$B.accbal' = B.accbal \oplus \{acc? \mapsto B.accbal(acc?) - amt?\}$
$H.accbal' = H.accbal$
$H.accounts' = H.accounts$
$B.accounts' = B.accounts$
$dom(H.accbal) = H.accounts$
$dom(H.accbal') = H.accounts'$
$dom(B.accbal) = B.accounts$
$dom(B.accbal') = B.accounts'$
$H.banktotal = \Sigma(B.accbal) + \Sigma(H.accbal)$
$H.banktotal' = \Sigma(B.accbal') + \Sigma(H.accbal')$

The LHS of expression (i) evaluates to the following:

$$
\begin{array}{|l}
\hline
BEState \wedge ((H.GWithdraw1D \fatsemi B.GWithdraw2D) \setminus (aux, aux')) \\
amt : \mathbb{N};\ acc : ACCOUNT \\
\hline
H.banktotal > amt? \\
acc? \in B.accounts \\
amt? < B.accbal(acc?) \\
H.banktotal' = H.banktotal - amt? \\
B.accbal' = B.accbal \oplus \{acc? \mapsto B.accbal(acc?) - amt?\} \\
H.accbal' = H.accbal \\
H.accounts' = H.accounts \\
B.accounts' = B.accounts \\
dom(H.accbal) = H.accounts \\
dom(H.accbal') = H.accounts' \\
dom(B.accbal) = B.accounts \\
dom(B.accbal') = B.accounts' \\
H.banktotal = \Sigma(B.accbal) + \Sigma(H.accbal) \\
H.banktotal' = \Sigma(B.accbal') + \Sigma(H.accbal') \\
\hline
\end{array}
$$

The two schemas are equivalent. Similarly, we can prove (ii), (iii) and (iv).

Semantic Atomicity Property

Proof Obligation :

Any partial semantic history H_p defined over a set of transactions $\mathbf{T}$ and containing one or more incomplete transactions of type *GTransfer*, *GAudit*, *GWithdraw*, *GDeposit* is a prefix of a complete semantic history defined over the same set of transactions.

Proof:

Suppose H_p has n incomplete transactions. We show how any incomplete transaction can be completed and the number of incomplete transactions reduced to

$n-1$. Repeating this process n times we get a complete semantic history. The partial semantic history H_p is therefore a prefix of the complete semantic history that we just generated. In the following paragraphs we show how the different types of incomplete transactions can complete.

Consider an incomplete *GTransfer* transaction. The history invariants impose no restriction of the execution of *GTrans*1*CompD*; the precondition of *GTrans*1*CompD* that checks for the validity of the account is also guaranteed to be satisfied (because the account is the one involved in *GTransfer*1*D* and so it is a valid account). Thus *GTrans*1*CompD* can execute and complete *GTransfer*. Similarly, we can show how to complete *GWithdraw*.

Consider an incomplete *GDeposit* transaction. The history invariants impose no restriction on *GDep*1*CompD*; *GDep*1*CompD* can execute if its preconditions are satisfied. The precondition of *GDep*1*CompD* may be violated if a *GWithdraw*1*D* executes between *GDeposit*1*D* and *GDep*1*CompD*. However, in this case *GWith*1*CompD* can execute and in turn establish the precondition of *GDep*1*CompD*.

Finally we show how *GAudit* transactions can complete by successfully executing *GAudit*2*D*. The history invariant imposes the restriction that *GAudit*2*D* cannot execute when integrity constraints are violated. Note that if *GTransfer*1*D*, *GDeposit*1*D*, *GWithdraw*1*D* violates the integrity constraints, the corresponding compensating steps can execute and establish the integrity constraints after which *GAudit*2*D* can execute.

Consistent Execution Property

Proof Obligation:

When a correct complete semantic history H_s is executed in a consistent state, the final state produced after the execution of the history is also consistent.

Proof:

First, let us consider the *GDeposit* transactions in the history. When a step of type $GDeposit1D$ executes, the auxiliary variable $H.tempDep1$ is incremented by the amount of money deposited by the step. When the corresponding step of type $GDeposit2D$ executes, the auxiliary variable $B.tempDep2$ is incremented by the same amount. Or, if the corresponding compensating step of type $GDep1CompD$ executes, the auxiliary variable $H.tempDep1Comp$ is incremented by the same amount. Thus for each increment in $H.tempDep1$ there is a corresponding increment in either $B.tempDep2$ or $H.tempDep1Comp$. Hence when all *GDeposit* steps complete, the following condition is satisfied: $H.tempDep1 = B.tempDep2 + H.tempDep1Comp$. Similarly, when all *GWithdraw* transactions complete the following condition is satisfied: $H.tempWith1 = B.tempWith2 + H.tempWith1Comp$. When all *GTransfer* transactions initiated at head office H complete, $H.tempXfer1 = H.tempXfer1Comp + B.tempXfer2$ holds. When all *GTransfer* transactions initiated at branch B complete, $B.tempXfer1 = B.tempXfer1Comp + H.tempXfer2$ holds. When all the above conditions hold, the generalized integrity constraint reduces to the original integrity constraint, which means that the database is restored to a consistent state.

Thus for this specification we do not have to take any extra measures for enforcing the history invariant specifying the consistent execution property. This history invariant can safely be eliminated from the refined specification.

Valid Successor Set Property

A possible successor set description for the banking example is given in Tables B.1 and B.2.

Table B.1: Successor Sets Description stored at Head Office H

SS of Type of Step	Types of Steps in the Successor Set
SS(GWithdraw1D)	GWithdraw1D, GDeposit1D, GTransfer1D, GTransfer2D, GTrans1CompD, GWith1CompD, GDep1CompD
SS(GDeposit1D)	GWithdraw1D, GDeposit1D, GTransfer1D, GTransfer2D, GTrans1CompD, GWith1CompD, GDep1CompD
SS(GTransfer1D)	GWithdraw1D, GDeposit1D, GTransfer1D, GTransfer2D, GTrans1CompD, GWith1CompD, GDep1CompD
SS(GAudit1D)	GAudit1D

Table B.2: Successor Sets Description stored at Branch B

SS of Type of Step	Types of Steps in the Successor Set
SS(GTransfer1D)	GWithdraw1D, GDeposit1D, GTransfer1D, GTransfer2D, GTrans1CompD, GWith1CompD, GDep1CompD

Proof Obligations:

(a) The refined specification satisfies the semantic atomicity property.

(b) Any complete correct successor set history generated by a successor set description given in Tables B.1 and B.2 is equivalent to some correct complete semantic history generated by the specification consisting of history invariants.

Proof of (a):

Consider a correct partial semantic history consisting of n incomplete transactions. We show how an incomplete transaction can complete and the number of incomplete transactions reduced to $n - 1$. Repeating this process n times we get a complete history. The correct partial semantic history is a prefix of the correct complete semantic history that has just been generated. In the following paragraphs we show how the different types of incomplete transactions can complete.

Consider an incomplete *GTransfer* transaction whose $GTransfer1D$ has committed at the branch. Since *GTrans1CompD* is in the successor set of *GTransfer1D* and the precondition of *GTrans1CompD* (which checks for the validity of the account) is always satisfied, *GTrans1CompD* can always execute and complete *GTransfer*. Next consider a *GTransfer* transaction whose *GTransfer1D* step has committed at the head office. Due to successor set constraints, *GTrans1CompD* can execute provided no other steps of type *GAudit1D* have executed after *GTransfer1D*. Also the successor set description will prevent *GAudit1D* from being executed after a *GTransfer1D*. This means that *GTrans1CompD* can always execute and complete *GTransfer*. Similarly we can prove that *GWithdraw* and *GDeposit* transactions will always complete.

Finally consider an incomplete *GAudit* transaction. The step of type *GAudit2D* can execute if the branch has no active *GTransfer* transaction. Alternatively, if the branch has active *GTransfer* transactions they can be completed by executing the corresponding *GTrans1CompD* after which *GAudit2D* can execute.

Proof of (b)

We denote the specifications with history invariants by S_1 and the refined specification by S_2. Consider any history H_s generated by the successor set mechanism.

Suppose H_s has some transactions whose first steps are committed in H and others whose first steps are committed in B. There can be two types of interleavings. Case (a) The head office has committed one or more steps of type *GAudit1D* and the branch has committed one or more steps of type *GTransfer1D*. In such a case, the only way the successor set description allows the history to be completed is the branch executing compensating steps of type *GTrans1CompD* for all incomplete *GTransfer* transactions. When all *GTransfer* transactions have completed, the steps of type *GAudit2D* can be processed. In this case the history generated from the successor set description is equivalent to a history in which the *GTransfer* transaction precedes the $GAudit$ transaction which is a correct history generated from S_1. Case (b) The head office has committed one or more steps which are not of type $GAudit1D$ and the branch has committed one or more steps of type $GTransfer1D$. In this case all possible interleavings do not involve steps of $GAudit$ and are permitted by the history invariants enforcing the sensitive transaction isolation property.

The proof for cases when H_s consists of transactions all of whose first steps are committed in B or H can be argued similarly.

Branch

$accounts : \mathbb{P}\,ACCOUNT;\ accbal : ACCOUNT \rightarrow\!\!\!\!\!+\ \mathbb{N}$

$\mathrm{dom}(accbal) = accounts$

INIT

$accbal = \varnothing$

LTransfer

$\Delta(accbal);\ amt? : \mathbb{N};\ acc1?, acc2? : ACCOUNT$

$\{acc1?, acc2?\} \subseteq accounts;\ amt? < accbal(acc1?)$

$accbal' = accbal \oplus \{acc1? \mapsto accbal(acc1?) - amt?,$
$\quad acc2? \mapsto accbal(acc2?) + amt?\}$

LAudit

$AccountBal! : ACCOUNT \rightarrow\!\!\!\!\!+\ \mathbb{N}$

$AccountBal! = accbal$

GWithdraw2

$\Delta(accbal);\ amt? : \mathbb{N};\ acc? : ACCOUNT$

$acc? \in accounts;\ amt? < accbal(acc?)$

$accbal' = accbal \oplus \{acc? \mapsto accbal(acc?) - amt?\}$

GDeposit2

$\Delta(accbal);\ amt? : \mathbb{N};\ acc? : ACCOUNT$

$acc? \in accounts$

$accbal' = accbal \oplus$
$\quad \{acc? \mapsto accbal(acc?) + amt?\}$

$GTransfer1 \mathrel{\widehat=} GWithdraw2;\ GTransfer2 \mathrel{\widehat=} GDeposit2;\ GAudit2 \mathrel{\widehat=} LAudit$

HeadOffice

$Branch[\textbf{remove}\ GWithdraw2, GDeposit2, GAudit2]$

$banktotal : \mathbb{N}$

INIT

$banktotal = 0$

LDeposit

$\Delta(accbal, banktotal);\ amt? : \mathbb{N};\ acc? : ACCOUNT$

$acc? \in accounts$

$accbal' = accbal \oplus$
$\quad \{acc1? \mapsto accbal(acc?) + amt?\}$

$banktotal' = banktotal + amt?$

LWithdraw

$\Delta(accbal, banktotal);\ amt? : \mathbb{N}$

$acc? : ACCOUNT$

$acc? \in accounts;\ amt? < accbal(acc?)$

$accbal' = accbal \oplus$
$\quad \{acc? \mapsto accbal(acc?) - amt?\}$

$banktotal' = banktotal - amt?$

GDeposit1

$\Delta(banktotal);\ amt? : \mathbb{N}$

$banktotal' = banktotal + amt?$

GWithdraw1

$\Delta(banktotal);\ amt? : \mathbb{N}$

$banktotal > amt?$

$banktotal' = banktotal - amt?$

GAudit1

$accbalance! : ACCOUNT \rightarrow\!\!\!\!\!+\ \mathbb{N};\ BankTotal! : \mathbb{N}$

$accbalance! = acctbalance;\ BankTotal! = banktotal$

Figure B.1: Initial Specification of Branch and Head Office

BankingEnterprise

$B : Branch$

$H : HeadOffice$

$H.banktotal = \Sigma(B.accbal) + \Sigma(H.accbal)$

INIT

$B.INIT$

$H.INIT$

SelectTwoBanks

$c1?, c2? :\downarrow Branch$

$c1? \in \{B, H\}$

$c2? \in \{B, H\}$

$c1? \neq c2?$

SelectBank

$c? :\downarrow Branch$

$c? \in \{B, H\}$

$LTransfer \mathrel{\widehat{=}} SelectBank \bullet c?.LTransfer$

$LAudit \mathrel{\widehat{=}} SelectBank \bullet c?.LAudit$

$LDeposit \mathrel{\widehat{=}} H.LDeposit$

$LWithdraw \mathrel{\widehat{=}} H.LWithdraw$

$GTransfer \mathrel{\widehat{=}} SelectTwoBanks \bullet (c1?.GTransfer1 \wedge c2?.GTransfer2)$

$GAudit \mathrel{\widehat{=}} H.GAudit1 \wedge B.GAudit2$

$GDeposit \mathrel{\widehat{=}} H.GDeposit1 \wedge B.GDeposit2$

$GWithdraw \mathrel{\widehat{=}} H.GWithdraw1 \wedge B.GWithdraw2$

Figure B.2: Initial Specification of the Banking Enterprise

BranchD

$$
\begin{array}{l}
accounts : \mathbb{P}\,ACCOUNT;\ accbal : ACCOUNT \pfun \mathbb{N} \\
tempXfer1, tempXfer2, tempXfer1Comp, tempDep2, tempWith2, tempAudit2 : \mathbb{N} \\
\hline
\mathrm{dom}(accbal) = accounts
\end{array}
$$

INIT

$$
\begin{array}{l}
accbal = \varnothing \\
tempAudit2 = 0 \\
tempDep2 = 0 \\
tempWith2 = 0 \\
tempXfer1 = 0 \\
tempXfer2 = 0 \\
tempXfer1Comp = 0
\end{array}
$$

GWithdraw2D

$$
\begin{array}{l}
\Delta(accbal, temp2);\ amt? : \mathbb{N};\ acc? : ACCOUNT \\
\hline
acc? \in accounts;\ amt? < accbal(acc?) \\
accbal' = accbal \oplus \{acc? \mapsto accbal(acc?) - amt?\} \\
tempWith2' = tempWith2 + amt?
\end{array}
$$

GTransfer1D

$$
\begin{array}{l}
\Delta(accbal, tempXfer1) \\
amt? : \mathbb{N};\ acc1? : ACCOUNT \\
\hline
acc1? \in accounts;\ amt? < accbal(acc1?) \\
accbal' = accbal \oplus \\
\qquad \{acc1? \mapsto accbal(acc1?) - amt?\} \\
tempXfer1' = tempXfer1 + amt?
\end{array}
$$

GTrans1CompD

$$
\begin{array}{l}
\Delta(accbal, tempXfer1Comp) \\
amt? : \mathbb{N};\ acc1? : ACCOUNT \\
\hline
acc1? \in accounts \\
accbal' = accbal \oplus \\
\qquad \{acc1? \mapsto accbal(acc1?) + amt?\} \\
tempXfer1Comp' = tempXfer1Comp + amt?
\end{array}
$$

LTransferD

$$
\begin{array}{l}
\Delta(accbal);\ amt? : \mathbb{N};\ acc1?, acc2? : ACCOUNT \\
\hline
\{acc1?, acc2?\} \subseteq accounts;\ amt? < accbal(acc1?) \\
accbal' = accbal \oplus \{acc1? \mapsto accbal(acc1?) - amt?, \\
\qquad acc2? \mapsto accbal(acc2?) + amt?\}
\end{array}
$$

LAuditD

$$
\begin{array}{l}
AccountBal! : ACCOUNT \pfun \mathbb{N} \\
\hline
AccountBal! = accbal
\end{array}
$$

GDeposit2D

$$
\begin{array}{l}
\Delta(accbal, tempDep2);\ amt? : \mathbb{N};\ acc? : ACCOUNT \\
\hline
acc? \in accounts \\
accbal' = accbal \oplus \\
\qquad \{acc? \mapsto accbal(acc?) + amt?\} \\
tempDep2' = tempDep2 + amt
\end{array}
$$

GTransfer2D

$$
\begin{array}{l}
\Delta(accbal, tempXfer2) \\
amt? : \mathbb{N};\ acc2? : ACCOUNT \\
\hline
acc2? \in accounts \\
accbal' = accbal \oplus \\
\qquad \{acc2? \mapsto accbal(acc2?) + amt?\} \\
tempXfer2' = tempXfer2 + amt?
\end{array}
$$

GAudit2D

$$
\begin{array}{l}
\Delta(tempAudit2);\ AccountBal! : ACCOUNT \pfun \mathbb{N} \\
\hline
AccountBal! = accbal \\
tempAudit2' = tempAudit2 + 1
\end{array}
$$

Figure B.3: Specification of Branch after Decomposition

HeadOfficeD

$Branch[\textbf{remove } GWithdraw2D,\ GDeposit2D,\ GAudit2D]$

$banktotal, tempAudit1, tempDep1, tempDep1Comp, tempWith1, tempWith1Comp : \mathbb{N}$

INIT

$tempDep1 = 0$

$tempDep1Comp = 0$

$tempWith1 = 0$

$tempWith1Comp = 0$

$tempAudit1 = 0$

$banktotal = 0$

LDepositD

$\Delta(accbal, banktotal);\ amt? : \mathbb{N}$

$acc? : ACCOUNT$

$acc? \in accounts$

$accbal' = accbal \oplus$
$\quad \{acc1? \mapsto accbal(acc?) + amt?\}$

$banktotal' = banktotal + amt?$

LWithdrawD

$\Delta(accbal, banktotal);\ amt? : \mathbb{N}$

$acc? : ACCOUNT$

$acc? \in accounts;\ amt? < accbal(acc?)$

$accbal' = accbal \oplus$
$\quad \{acc? \mapsto accbal(acc?) - amt?\}$

$banktotal' = banktotal - amt?$

GDeposit1D

$\Delta(banktotal, tempDep1);\ amt? : \mathbb{N}$

$banktotal' = banktotal + amt?$

$tempDep1' = tempDep1 + amt?$

GDep1CompD

$\Delta(banktotal, tempDep1Comp);\ amt? : \mathbb{N}$

$banktotal > amt?$

$banktotal' = banktotal - amt?$

$tempDep1Comp' = tempDep1Comp + amt?$

GWithdraw1D

$\Delta(banktotal, tempWith1);\ amt? : \mathbb{N}$

$banktotal > amt?$

$banktotal' = banktotal - amt?$

$tempWith1' = tempWith1 + amt?$

GAudit1D

$\Delta(tempAudit1)$

$accbalance! : ACCOUNT \nrightarrow \mathbb{N}$

$BankTotal! : \mathbb{N}$

$tempAudit1' = tempAudit1 + 1$

$accbalance! = acctbalance$

$BankTotal! = banktotal$

GWith1CompD

$\Delta(banktotal, tempWith1Comp);\ amt? : \mathbb{N}$

$banktotal' = banktotal + amt?$

$tempWith1Comp' = tempWith1Comp + amt?$

Figure B.4: Specification of Head Office after Decomposition

BankingEnterpriseD

$$B : BranchD$$
$$H : HeadOfficeD$$

$$\begin{aligned} banktotal = {} & \Sigma(B.accbal) + \Sigma(H.accbal) \\ & +H.tempDep1 - B.tempDep2 - H.tempDep1Comp \\ & -H.tempWith1 + B.tempWith2 + H.tempWith1Comp \\ & +H.tempXfer1 - B.tempXfer2 - H.tempXfer1Comp \\ & +B.tempXfer1 - H.tempXfer2 - B.tempXfer1Comp \end{aligned}$$

INIT

$$B.INIT$$
$$H.INIT$$

SelectBank

$$c? :\downarrow Branch$$
$$c? \in \{B, H\}$$

$$LTransferD \mathrel{\widehat=} SelectBank \bullet c?.LTransferD$$
$$LAuditD \mathrel{\widehat=} SelectBank \bullet c?.LAuditD$$
$$LDepositD \mathrel{\widehat=} H.LDepositD$$
$$LWithdrawD \mathrel{\widehat=} H.LWithdrawD$$
$$GTransfer1D \mathrel{\widehat=} SelectBank \bullet c?.GTransfer1D$$
$$GTransfer2D \mathrel{\widehat=} SelectBank \bullet c?.GTransfer2D$$
$$GAudit1D \mathrel{\widehat=} H.GAudit1D$$
$$GAudit2D \mathrel{\widehat=} B.GAudit2D$$
$$GDeposit1D \mathrel{\widehat=} H.GDeposit1D$$
$$GDeposit2D \mathrel{\widehat=} B.GDeposit2D$$
$$GWithdraw1D \mathrel{\widehat=} H.GWithdraw1D$$
$$GWithdraw2D \mathrel{\widehat=} B.GWithdraw2D$$

$$\begin{aligned} & \Box((H.tempDep1 = B.tempDep2 + H.tempDep1Comp) \wedge (H.tempWith1 = B.tempWith2 + \\ & \quad H.tempWith1Comp) \wedge (H.tempXfer1 = B.tempXfer2 + H.tempXfer1Comp) \wedge (B.tempXfer1 = \\ & \quad H.tempXfer2 + B.tempXfer1Comp) \Rightarrow (H.banktotal = \Sigma B.accbal + \Sigma H.accbal)) \\ & \Box((H.banktotal \neq \Sigma H.accbal + \Sigma B.accbal) \Rightarrow ((\bigcirc\, \mathbf{op} \neq GAudit1D) \wedge (\bigcirc\, \mathbf{op} \neq GAudit2D)) \\ & \Box((H.tempAudit1 \neq B.tempAudit2) \Rightarrow ((\bigcirc\, \mathbf{op} \neq GDeposit2D) \wedge (\bigcirc\, \mathbf{op} \neq GWithdraw2D) \\ & \quad \wedge (\bigcirc\, \mathbf{op} \neq GTransfer2D))) \end{aligned}$$

Figure B.5: Specification of Banking Enterprise after Decomposition

Appendix C

Example Decomposition of Multilevel Transactions

We take a Mission Database as an example of a multilevel secure database. The Mission Database has a set of resources which can be assigned to threats. The status of each resource is *Busy* or *Idle* depending on whether or not the resource is assigned to some threat – this information is stored in the database object *Status*. The information about which resource is assigned to which threat is stored in the object *Assign*. There are three integrity constraints specified on these objects: (i) each resource can have only one status, (ii) each resource can be assigned to at most one threat, and (iii) the set of resources assigned to threats equals the set of resources with status *Busy*.

The three types of transactions in the Mission Database are *Respond*, *Cancel* and *Report*. *Respond* assigns some idle resource to a threat, *Cancel* cancels the assignment of a resource to a threat, and *Respond* prints the status of each resource and the information about resource to threat assignment. *Respond* has a precondition that the status of resource $r?$ which is to be assigned must be *Idle*. The postcondition of *Respond* is that the status of $r?$ is changed to *Busy* and the resource $r?$ is assigned to threat $t?$. *Cancel* has a precondition that the resource whose assignment is to be canceled must be assigned to some threat. The postcondition of *Cancel* deassigns the resource from the function *Assign* and changes the status of the resource to *Idle*. *Report* transaction has no preconditions and merely produces the state components

Mission

$Assign : Resource \nrightarrow Threat$

$Status : Resource \nrightarrow Status$

$\mathrm{dom}(Status \rhd \{Busy\}) = \mathrm{dom}\ Assign$

INIT

$Assign = \varnothing$

$\forall\, r : \mathrm{dom}\ Status \bullet Status(r) = Idle$

Respond

$\Delta(Status, Assign)$

$t? : Threat$

$r? : Resource$

$Status(r?) = Idle$

$Status' = Status \oplus \{r? \mapsto Busy\}$

$Assign' = Assign \cup \{r? \mapsto t?\}$

Cancel

$\Delta Mission;\ r? : Resource$

$r? \in \mathrm{dom}\ Assign$

$Status' = Status \oplus \{r? \mapsto Idle\}$

$Assign' = \{r?\} \lhd Assign$

Report

$currentstatus! : Resource \nrightarrow Status$

$currentassignments! : Resource \nrightarrow Threat$

$currentstatus! = Status$

$currentassignments! = Assign$

Figure C.1: Initial Specification of the Mission Database

Status and *Assign* as outputs.

The Object Z specification of this Mission Database appears in figure C.1. The class *Mission* specifies the objects, integrity constraints and transactions of the Mission Database. The state schema which is nameless lists the objects and the integrity constraints; the operation schemas specify the transactions.

C.1 Decomposition of Multilevel Transactions in the Mission Database

The class *DecomposedMission* which lists the database objects, auxiliary variables, generalized integrity constraints and the steps of the decomposed transactions is shown in figure C.2. The *Respond* transaction is decomposed into a secret step $R1$

and a top secret step $R2$. Similarly the *Cancel* transaction is decomposed into steps $C1$ and $C2$ where level of $C1$ is secret and that of $C2$ is top secret. For each of these transactions, the secret level step must be executed before the top secret level step is executed; thus *Respond* and *Cancel* transactions are in the canonical form. The *Report* transaction is composed of a single top secret step.

After a step has been executed, the original integrity constraints may no longer be satisfied. For this reason, we have generalized the integrity constraints; these generalized constraints are satisfied before and after the execution of each step [AJR95]. The generalized integrity constraint for the *Mission* database is given in the constraint part of the state schema of class *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 step $R1$, is included in the bag $R1ax$ in step $R1$. Finally when the resource is assigned to the threat in step $R2$, 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.

Note that additional preconditions are introduced in steps $R2$,$C1$, *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.

C.2 Properties of the Decomposition

Composition Property

Proof Obligations:

(i) $MissionState \land (R1 \fatsemi R2) \setminus (aux, aux') \Leftrightarrow Respond$

(ii) $MissionState \land (C1 \fatsemi C2) \setminus (aux, aux') \Leftrightarrow Cancel$

where $MissionState$ denotes the nameless schema in Figure C.1, and $aux = R1ax, R2ax, C1ax, C2ax$, and $aux' = R1ax', R2ax', C1ax', C2ax'$.

Proof of (i):

The left hand side of (i) evaluates to

$$
\begin{array}{|l}
\hline
MissionState \land (R1 \fatsemi R2) \setminus (R1ax, R1ax', C1ax, C1ax', R2ax, R2ax', C2ax, C2ax') \\
\Delta(Status, Assign) \\
t? : Threat \\
r? : Resource \\
\hline
Status(r?) = Idle \\
Status' = Status \oplus \{r? \mapsto Busy\} \\
Assign' = Assign \cup \{r? \mapsto t?\} \\
\mathrm{dom}(Status' \rhd \{Busy\}) \cup \mathrm{dom}(C1ax' \uplus C2ax') \\
\qquad = \mathrm{dom}\ Assign' \cup \mathrm{dom}(R1ax' \uplus R2ax') \\
\mathrm{dom}(Status \rhd \{Busy\}) \cup \mathrm{dom}(C1ax \uplus C2ax) \\
\qquad = \mathrm{dom}\ Assign \cup \mathrm{dom}(R1ax \uplus R2ax) \\
\hline
\end{array}
$$

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

Using similar arguments we can prove (ii).

Consistent Execution Property

Proof Obligation:

For any correct complete semantic history H, if the initial state satisfies the integrity constraints listed in the nameless schema of class *Mission* (figure C.1), then the final state also satisfies 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 H. 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 final state of H. Note that the variable $R1ax$ is updated in step $R1$ and variable $R2ax$ is updated similarly in step $R2$. Hence, when all the *Reserve* transactions have completed execution, the variables $R1ax$ and $R2ax$ are changed in the same way. Therefore, in the final state of H, $\mathrm{dom}((\widehat{R1ax} \uplus \widehat{R2ax}) = \mathrm{dom}((\widetilde{R1ax} \uplus \widetilde{R2ax})$

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

Thus the following holds in the final state of H: $\mathrm{dom}((\widehat{R1ax} \uplus \widehat{R2ax}) = \varnothing \wedge (\widehat{C1ax} \uplus \widehat{C2ax})) = \varnothing$. 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.

Successful Execution 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 defined over the same set of transactions.

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 steps. We refer to a transaction that has completed one or more but not all the steps 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.

(Note that as stated in Section 6.5 a slight change in requirements – a threat can be assigned only one resource – can cause the application to lose the successful execution property.)

Complete Execution 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:

Since the example application has the successful execution property, it also has the complete execution property.

DecomposedMission

$Status : Resource \mathrel{+\!\!\!\!\to} Status$

$R1ax, C1ax : \mathrm{bag}\ Resource$

$Assign : Resource \mathrel{+\!\!\!\!\to} Threat$

$R2ax, C2ax : \mathrm{bag}\ Resource$

$\mathrm{dom}(Status \rhd \{Busy\}) \cup \mathrm{dom}(C1ax \uplus C2ax) = \mathrm{dom}\ Assign \cup \mathrm{dom}(R1ax \uplus R2ax)$

INIT

$\forall\, r : \mathrm{dom}\ Status \bullet Status(r) = Idle$

$Assign = \varnothing$

$R1ax = \varnothing;\ C1ax = \varnothing$

$R2ax = \varnothing;\ C2ax = \varnothing$

Report1

$currentstatus! : Resource \mathrel{+\!\!\!\!\to} Status$

$currentassignments! : Resource \mathrel{+\!\!\!\!\to} Threat$

$R1ax = R2ax;\ C1ax = C2ax$

$currentstatus! = Status$

$currentassignments! = Assign$

R1

$\Delta(Status, R1ax)$

$r? : Resource$

$Status(r?) = Idle$

$Status' = Status \oplus \{r? \mapsto Busy\}$

$R1ax' = R1ax \uplus [\![r?]\!]$

R2

$\Delta(Assign, R2ax)$

$t? : Threat;\ r? : Resource$

$r? \notin \mathrm{dom}\ Assign$

$Assign' = Assign \cup \{r? \mapsto t?\}$

$R2ax' = R2ax \uplus [\![r?]\!]$

C1

$\Delta(Status, C1ax)$

$r? : Resource$

$Status(r?) = Busy$

$Status' = Status \oplus \{r? \mapsto Idle\}$

$C1ax' = C1ax \uplus [\![r?]\!]$

C2

$\Delta(Assign, C2ax);\ r? : Resource$

$r? \in \mathrm{dom}\ Assign$

$Assign' = \{r?\} \ntriangleleft Assign$

$C2ax' = C2ax \uplus [\![r?]\!]$

Figure C.2: A Valid Decomposition for the Mission Database

Appendix D

Example AMN Specification

D.1 AMN Specification for the Hotel Example

The AMN specification corresponding to the Hotel Example specified in Chapter 7 is given below. The B-Toolkit checks to ensure that the format is correct and there are no inconsistencies.

MACHINE
  $HotelD$
SETS
  $GUEST$ **;**
  $ROOM$ **;**
  $STATUS = \{ Available , Taken \}$
CONSTANTS
  $total$
PROPERTIES
  $total = 100$
VARIABLES
  res ,
  $underway$,
  $acquired$,
  RM ,
  ST
INVARIANT
  $ST \in ROOM \nrightarrow STATUS \wedge$
  $RM \in GUEST \rightarrowtail\!\!\!\!\rightarrow ROOM \wedge$
  $res \in \mathbb{N} \wedge$
  $underway \in \mathbb{N} \wedge$
  $acquired \in \mathbb{P} (ROOM) \wedge$
  $\mathsf{card} (RM) + underway = res \wedge$

$\mathsf{dom}\ (\ ST \rhd \{\ Taken\ \}\ ) = \mathsf{ran}\ (\ RM\ ) \cup acquired \wedge$
$\mathsf{ran}\ (\ RM\ ) \cap acquired = \varnothing$

INITIALISATION

$res\ ,\ underway\ ,\ acquired\ ,\ RM\ ,\ ST := 0\ ,\ 0\ ,\ \varnothing\ ,\ \varnothing\ ,\ ROOM \times \{\ Available\ \}$

OPERATIONS

R1

$R1 \quad ==$
PRE $res < total$
THEN
$res := res + 1 \ \|$
$underway := underway + 1$
END ;

R2

$R2\ (\ room\_inp\ ) \quad ==$
PRE $room\_inp \in ROOM \wedge underway > 0 \wedge ST\ (\ room\_inp\ ) = Available$
THEN
$ST := ST \lhd\!\!-\ \{\ room\_inp \mapsto Taken\ \} \ \|$
$acquired := acquired \cup \{\ room\_inp\ \}$
END ;

R3

$R3\ (\ room\_inp\ ,\ guest\_inp\ ) \quad ==$
PRE $room\_inp \in ROOM \wedge guest\_inp \in GUEST \wedge guest\_inp \notin \mathsf{dom}\ (\ RM\ )$
$\wedge\ underway > 0 \wedge room\_inp \in acquired$
THEN
$RM := RM \cup \{\ guest\_inp \mapsto room\_inp\ \} \ \|$
$acquired := acquired - \{\ room\_inp\ \} \ \|$
$underway := underway - 1$
END ;

CompR1

$CompR1 \quad ==$
PRE $res > 0 \wedge underway > 0$ **THEN**
$res := res - 1 \ \|$
$underway := underway - 1$
END ;

CompR2

$CompR2\ (\ room\_inp\ ) \quad ==$
PRE $room\_inp \in ROOM \wedge room\_inp \in acquired$ **THEN**
$acquired := acquired - \{\ room\_inp\ \} \ \|$

$ST := ST \mathbin{\lhd\!\!-} \{ room\_inp \mapsto Available \}$
END ;

Cancel

$Cancel (guest\_inp) \quad ==$
PRE $guest\_inp \in \mathsf{dom} (RM)$
THEN
$ST (RM (guest\_inp)) := Available \;\|$
$RM := \{ guest\_inp \} \mathbin{\lhd\!\!\!-} RM \;\|$
$res := res - 1$
END ;

Report

$status\_out , assign\_out \longleftarrow Report \quad ==$
PRE $acquired = \varnothing$ **THEN**
$status\_out , assign\_out := ST , RM$
END

END

D.2 Proof Obligations for the Hotel Example

After checking the consistency of the specification, we invoke the Proof Obligation Generator. The Proof Obligation Generator automatically generates the following proof obligations.

Proof Obligations for HotelD.mch

Initialisation . 1

$cst (HotelD) \wedge ctx (HotelD)$
$\Rightarrow$
$ROOM \times \{ Available \} \in ROOM \to STATUS$

Initialisation . 2

$cst (HotelD) \wedge ctx (HotelD)$
$\Rightarrow$
$\varnothing \in GUEST \rightarrowtail\!\!\!\!\!\!\!\!\;\;+\;\; ROOM$

Initialisation . 3

$cst (HotelD) \wedge ctx (HotelD)$

$\Rightarrow$
$\mathit{0} \in \mathbb{N}$

$\underline{\mathit{Initialisation\ .\ 4}}$

$\mathit{cst}\ (\ \mathit{HotelD}\ ) \wedge \mathit{ctx}\ (\ \mathit{HotelD}\ )$
$\Rightarrow$
$\varnothing \in \mathbb{P}\ (\ \mathit{ROOM}\ )$

$\underline{\mathit{Initialisation\ .\ 5}}$

$\mathit{cst}\ (\ \mathit{HotelD}\ ) \wedge \mathit{ctx}\ (\ \mathit{HotelD}\ )$
$\Rightarrow$
$\mathsf{card}\ (\ \varnothing\ ) + \mathit{0} = \mathit{0}$

$\underline{\mathit{Initialisation\ .\ 6}}$

$\mathit{cst}\ (\ \mathit{HotelD}\ ) \wedge \mathit{ctx}\ (\ \mathit{HotelD}\ )$
$\Rightarrow$
$\mathsf{dom}\ (\ \mathit{ROOM}\ \times \{\ \mathit{Available}\ \} \rhd \{\ \mathit{Taken}\ \}\ ) = \mathsf{ran}\ (\ \varnothing\ ) \cup \varnothing$

$\underline{\mathit{Initialisation\ .\ 7}}$

$\mathit{cst}\ (\ \mathit{HotelD}\ ) \wedge \mathit{ctx}\ (\ \mathit{HotelD}\ )$
$\Rightarrow$
$\mathsf{ran}\ (\ \varnothing\ ) \cap \varnothing = \varnothing$

$\underline{\mathit{R1\ .\ 1}}$

$\mathit{cst}\ (\ \mathit{HotelD}\ ) \wedge \mathit{ctx}\ (\ \mathit{HotelD}\ ) \wedge$
$\mathit{inv}\ (\ \mathit{HotelD}\ ) \wedge \mathit{asn}\ (\ \mathit{HotelD}\ ) \wedge$
$\mathit{pre}\ (\ \mathit{R1}\ )$
$\Rightarrow$
$\mathit{res} + \mathit{1} \in \mathbb{N}$

$\underline{\mathit{R1\ .\ 2}}$

$\mathit{cst}\ (\ \mathit{HotelD}\ ) \wedge \mathit{ctx}\ (\ \mathit{HotelD}\ ) \wedge$
$\mathit{inv}\ (\ \mathit{HotelD}\ ) \wedge \mathit{asn}\ (\ \mathit{HotelD}\ ) \wedge$
$\mathit{pre}\ (\ \mathit{R1}\ )$
$\Rightarrow$
$\mathit{underway} + \mathit{1} \in \mathbb{N}$

$\underline{\mathit{R1\ .\ 3}}$

$\mathit{cst}\ (\ \mathit{HotelD}\ ) \wedge \mathit{ctx}\ (\ \mathit{HotelD}\ ) \wedge$
$\mathit{inv}\ (\ \mathit{HotelD}\ ) \wedge \mathit{asn}\ (\ \mathit{HotelD}\ ) \wedge$
$\mathit{pre}\ (\ \mathit{R1}\ )$
$\Rightarrow$
$\mathsf{card}\ (\ \mathit{RM}\ ) + (\ \mathit{underway} + \mathit{1}\ ) = \mathit{res} + \mathit{1}$

$\underline{\mathit{R2\ .\ 1}}$

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ R2\ )$
$\Rightarrow$
$ST \mathbin{\lhd\!\!\!+} \{\ room\_inp \mapsto Taken\ \} \in ROOM \mathrel{+\!\!\!\!\rightarrow} STATUS$

R2 . 2

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ R2\ )$
$\Rightarrow$
$acquired \cup \{\ room\_inp\ \} \in \mathbb{P}\ (\ ROOM\ )$

R2 . 3

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ R2\ )$
$\Rightarrow$
$\mathsf{dom}\ (\ ST \mathbin{\lhd\!\!\!+} \{\ room\_inp \mapsto Taken\ \} \rhd \{\ Taken\ \}\ ) = \mathsf{ran}\ (\ RM\ ) \cup (\ acquired \cup \{\ room\_inp\ \}\ )$

R2 . 4

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ R2\ )$
$\Rightarrow$
$\mathsf{ran}\ (\ RM\ ) \cap (\ acquired \cup \{\ room\_inp\ \}\ ) = \varnothing$

R3 . 1

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ R3\ )$
$\Rightarrow$
$RM \cup \{\ guest\_inp \mapsto room\_inp\ \} \in GUEST \mathrel{\rightarrowtail\!\!\!\!\!+} ROOM$

R3 . 2

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ R3\ )$
$\Rightarrow$
$underway - 1 \in \mathbb{N}$

R3 . 3

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ R3\ )$

$\Rightarrow$
$acquired - \{ room\_inp \} \in \mathbb{P} (ROOM)$

<u>*R3 . 4*</u>

$cst (HotelD) \wedge ctx (HotelD) \wedge$
$inv (HotelD) \wedge asn (HotelD) \wedge$
$pre (R3)$
$\Rightarrow$
$\mathsf{card} (RM \cup \{ guest\_inp \mapsto room\_inp \}) + (underway - 1) = res$

<u>*R3 . 5*</u>

$cst (HotelD) \wedge ctx (HotelD) \wedge$
$inv (HotelD) \wedge asn (HotelD) \wedge$
$pre (R3)$
$\Rightarrow$
$\mathsf{dom} (ST \rhd \{ Taken \}) = \mathsf{ran} (RM \cup \{ guest\_inp \mapsto room\_inp \}) \cup acquired - \{ room\_inp \}$

<u>*R3 . 6*</u>

$cst (HotelD) \wedge ctx (HotelD) \wedge$
$inv (HotelD) \wedge asn (HotelD) \wedge$
$pre (R3)$
$\Rightarrow$
$\mathsf{ran} (RM \cup \{ guest\_inp \mapsto room\_inp \}) \cap acquired - \{ room\_inp \} = \varnothing$

<u>*CompR1 . 1*</u>

$cst (HotelD) \wedge ctx (HotelD) \wedge$
$inv (HotelD) \wedge asn (HotelD) \wedge$
$pre (CompR1)$
$\Rightarrow$
$res - 1 \in \mathbb{N}$

<u>*CompR1 . 2*</u>

$cst (HotelD) \wedge ctx (HotelD) \wedge$
$inv (HotelD) \wedge asn (HotelD) \wedge$
$pre (CompR1)$
$\Rightarrow$
$underway - 1 \in \mathbb{N}$

<u>*CompR1 . 3*</u>

$cst (HotelD) \wedge ctx (HotelD) \wedge$
$inv (HotelD) \wedge asn (HotelD) \wedge$
$pre (CompR1)$
$\Rightarrow$
$\mathsf{card} (RM) + (underway - 1) = res - 1$

$\underline{CompR2\ .\ 1}$

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ CompR2\ )$
$\Rightarrow$
$ST \lhd\!\!\!\!+ \{\ room\_inp \mapsto Available\ \} \in ROOM \nrightarrow STATUS$

$\underline{CompR2\ .\ 2}$

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ CompR2\ )$
$\Rightarrow$
$acquired - \{\ room\_inp\ \} \in \mathbb{P}\ (\ ROOM\ )$

$\underline{CompR2\ .\ 3}$

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ CompR2\ )$
$\Rightarrow$
$\mathsf{dom}\ (\ ST \lhd\!\!\!\!+ \{\ room\_inp \mapsto Available\ \} \rhd \{\ Taken\ \}\ ) = \mathsf{ran}\ (\ RM\ ) \cup acquired - \{\ room\_inp\ \}$

$\underline{CompR2\ .\ 4}$

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ CompR2\ )$
$\Rightarrow$
$\mathsf{ran}\ (\ RM\ ) \cap acquired - \{\ room\_inp\ \} = \varnothing$

$\underline{Cancel\ .\ 1}$

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ Cancel\ )$
$\Rightarrow$
$ST \lhd\!\!\!\!+ \{\ RM\ (\ guest\_inp\ ) \mapsto Available\ \} \in ROOM \nrightarrow STATUS$

$\underline{Cancel\ .\ 2}$

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$
$pre\ (\ Cancel\ )$
$\Rightarrow$
$\{\ guest\_inp\ \} \trianglelefteq\!\!\!\!- RM \in GUEST \rightarrowtail\!\!\!\!\!\nrightarrow ROOM$

$\underline{Cancel\ .\ 3}$

$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge$
$inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge$

$$pre\ (\ Cancel\ ) \Rightarrow res - 1 \in \mathbb{N}$$

<u>*Cancel . 4*</u>

$$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge pre\ (\ Cancel\ ) \Rightarrow \mathsf{card}\ (\ \{\ guest\_inp\ \} \lhd RM\ ) + underway = res - 1$$

<u>*Cancel . 5*</u>

$$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge pre\ (\ Cancel\ ) \Rightarrow \mathsf{dom}\ (\ ST \mathbin{\lhd\!\!\!+} \{\ RM\ (\ guest\_inp\ ) \mapsto Available\ \} \rhd \{\ Taken\ \}\ ) = \mathsf{ran}\ (\ \{\ guest\_inp\ \} \lhd RM\ ) \cup acquired$$

<u>*Cancel . 6*</u>

$$cst\ (\ HotelD\ ) \wedge ctx\ (\ HotelD\ ) \wedge inv\ (\ HotelD\ ) \wedge asn\ (\ HotelD\ ) \wedge pre\ (\ Cancel\ ) \Rightarrow \mathsf{ran}\ (\ \{\ guest\_inp\ \} \lhd RM\ ) \cap acquired = \varnothing$$

<u>*Context . 1*</u>

$$cst\ (\ HotelD\ ) \wedge STATUS = \{\ Available\ ,\ Taken\ \} \wedge \mathsf{card}\ (\ STATUS\ ) = 2 \Rightarrow \exists\ GUEST\ .\ (\ \mathsf{card}\ (\ GUEST\ ) \in \mathbb{N}_1\ )$$

<u>*Context . 2*</u>

$$cst\ (\ HotelD\ ) \wedge STATUS = \{\ Available\ ,\ Taken\ \} \wedge \mathsf{card}\ (\ STATUS\ ) = 2 \Rightarrow \exists\ ROOM\ .\ (\ \mathsf{card}\ (\ ROOM\ ) \in \mathbb{N}_1\ )$$

D.3 Example Proofs Discharged Using AutoProver

Some proof obligations were automatically discharged by the AutoProver. Below we give few examples of the proofs done by the AutoProver.

Proofs (Level 1) for HotelD.mch

<u>*Initialisation . 1*</u>

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$STATUS = \{\ Available\ ,\ Taken\ \}$	2,Props.4
4	$ROOM \subseteq ROOM$	Law.1
5	$Available \in \{\ Available\ ,\ Taken\ \}$	Law.2
6	$Available \in STATUS$	5,3
7	$ROOM \times \{\ Available\ \} \in ROOM \nrightarrow STATUS$	4,6,Law.3
8	QED	DED

<u>*Initialisation . 2*</u>

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$\varnothing \in GUEST ⤔ ROOM$	Law.4
4	QED	DED

<u>*Initialisation . 3*</u>

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$0 \in \mathbb{N}$	Law.5
4	QED	DED

<u>*Initialisation . 4*</u>

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$\varnothing \subseteq ROOM$	Law.6
4	$\varnothing \in \mathbb{P}\ (\ ROOM\ )$	Law.7
5	QED	DED

<u>*Initialisation . 7*</u>

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$\mathsf{ran}\ (\ \varnothing\ ) \cap \varnothing \subseteq \varnothing$	Law.8
4	$\mathsf{ran}\ (\ \varnothing\ ) \cap \varnothing = \varnothing$	Law.9
5	QED	DED

R1 . 1

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$inv\ (\ HotelD\ )$	HYP
4	$asn\ (\ HotelD\ )$	HYP
5	$\quad res \in \mathbb{N}$	3,HypExp.3
6	$\quad pre\ (\ R1\ )$	HYP
7	$\qquad 1 \in \mathbb{N}$	Law.5
8	$\qquad res + 1 \in \mathbb{N}$	5,7,Law.10
9	QED	DED

R1 . 2

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$inv\ (\ HotelD\ )$	HYP
4	$asn\ (\ HotelD\ )$	HYP
5	$\quad underway \in \mathbb{N}$	3,HypExp.4
6	$\quad pre\ (\ R1\ )$	HYP
7	$\qquad 1 \in \mathbb{N}$	Law.5
8	$\qquad underway + 1 \in \mathbb{N}$	5,7,Law.10
9	QED	DED

R2 . 1

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$inv\ (\ HotelD\ )$	HYP
4	$asn\ (\ HotelD\ )$	HYP
5	$\quad STATUS = \{\ Available\ ,\ Taken\ \}$	2,Props.4
6	$\quad ST \in ROOM \nrightarrow STATUS$	3,HypExp.1
7	$\quad pre\ (\ R2\ )$	HYP
8	$\qquad room\_inp \in ROOM$	7,HypExp.10
9	$\qquad Taken \in \{\ Available\ ,\ Taken\ \}$	Law.11
10	$\qquad Taken \in STATUS$	9,5
11	$\qquad ST \mathbin{\lhd\!\!+} \{\ room\_inp \mapsto Taken\ \} \in ROOM \nrightarrow STATUS$	6,8,10,Law.12
12	QED	DED

R3 . 2

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$inv\ (\ HotelD\ )$	HYP
4	$asn\ (\ HotelD\ )$	HYP
5	$\quad underway \in \mathbb{N}$	3,HypExp.4
6	$\quad pre\ (\ R3\ )$	HYP
7	$\qquad 0 < underway$	6,HypExp.16
8	$\qquad 1 \in \mathbb{N}$	Law.5
9	$\qquad 1 = 1$	EQL

10	$0 \leq 0$	Law.13
11	$1 \leq underway$	5,7,Law.14
12	$underway - 1 \in \mathbb{N}$	5,8,11,Law.15
13	QED	DED

$\underline{CompR1\ .\ 1}$

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$inv\ (\ HotelD\ )$	HYP
4	$asn\ (\ HotelD\ )$	HYP
5	$res \in \mathbb{N}$	3,HypExp.3
6	$pre\ (\ CompR1\ )$	HYP
7	$0 < res$	6,HypExp.18
8	$1 \in \mathbb{N}$	Law.5
9	$1 = 1$	EQL
10	$0 \leq 0$	Law.13
11	$1 \leq res$	5,7,Law.14
12	$res - 1 \in \mathbb{N}$	5,8,11,Law.15
13	QED	DED

$\underline{CompR1\ .\ 2}$

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$inv\ (\ HotelD\ )$	HYP
4	$asn\ (\ HotelD\ )$	HYP
5	$underway \in \mathbb{N}$	3,HypExp.4
6	$pre\ (\ CompR1\ )$	HYP
7	$0 < underway$	6,HypExp.19
8	$1 \in \mathbb{N}$	Law.5
9	$1 = 1$	EQL
10	$0 \leq 0$	Law.13
11	$1 \leq underway$	5,7,Law.14
12	$underway - 1 \in \mathbb{N}$	5,8,11,Law.15
13	QED	DED

$\underline{CompR2\ .\ 1}$

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$inv\ (\ HotelD\ )$	HYP
4	$asn\ (\ HotelD\ )$	HYP
5	$STATUS = \{\ Available\ ,\ Taken\ \}$	2,Props.4
6	$ST \in ROOM \nrightarrow STATUS$	3,HypExp.1
7	$pre\ (\ CompR2\ )$	HYP
8	$room\_inp \in ROOM$	7,HypExp.20
9	$Available \in \{\ Available\ ,\ Taken\ \}$	Law.2
10	$Available \in STATUS$	9,5

11	$\quad ST \mathbin{\lhd\!\!\!+} \{ room\_inp \mapsto Available \} \in ROOM \nrightarrow STATUS$	6,8,10,Law.12
12	QED	DED

<u>*CompR2 . 4*</u>

1	$cst (HotelD)$	HYP
2	$ctx (HotelD)$	HYP
3	$inv (HotelD)$	HYP
4	$asn (HotelD)$	HYP
5	$\quad \mathsf{ran} (RM) \cap acquired = \varnothing$	3,HypExp.8
6	$\quad pre (CompR2)$	HYP
7	$\qquad \mathsf{ran} (RM) \cap acquired - \{ room\_inp \} = \varnothing$	5,Law.16
8	QED	DED

<u>*Context . 1*</u>

1	$cst (HotelD)$	HYP
2	$STATUS = \{ Available , Taken \}$	HYP
3	$\mathsf{card} (STATUS) = 2$	HYP
4	$\quad \exists GUEST . (\mathsf{card} (GUEST) \in \mathbb{N}_1)$	Law.17
5	QED	DED

<u>*Context . 2*</u>

1	$cst (HotelD)$	HYP
2	$STATUS = \{ Available , Taken \}$	HYP
3	$\mathsf{card} (STATUS) = 2$	HYP
4	$\quad \exists ROOM . (\mathsf{card} (ROOM) \in \mathbb{N}_1)$	Law.17
5	QED	DED

Laws (Level 1) for HotelD.mch

<u>*Props . 4*</u>

$ctx (HotelD)$
$\Rightarrow$
$STATUS = \{ Available , Taken \}$

<u>*HypExp . 3*</u>

$inv (HotelD)$
$\Rightarrow$
$res \in \mathbb{N}$

<u>*HypExp . 4*</u>

$inv (HotelD)$
$\Rightarrow$
$underway \in \mathbb{N}$

<u>*HypExp . 1*</u>

$inv (HotelD)$
$\Rightarrow$
$ST \in ROOM \nrightarrow STATUS$

<u>*HypExp . 10*</u>

$pre\ (\ R2\ )$
$\Rightarrow$
$room\_inp \in ROOM$

<u>*HypExp . 16*</u>

$pre\ (\ R3\ )$
$\Rightarrow$
$0 < underway$

<u>*HypExp . 18*</u>

$pre\ (\ CompR1\ )$
$\Rightarrow$
$0 < res$

<u>*HypExp . 19*</u>

$pre\ (\ CompR1\ )$
$\Rightarrow$
$0 < underway$

<u>*HypExp . 20*</u>

$pre\ (\ CompR2\ )$
$\Rightarrow$
$room\_inp \in ROOM$

<u>*HypExp . 8*</u>

$inv\ (\ HotelD\ )$
$\Rightarrow$
$\mathsf{ran}\ (\ RM\ ) \cap acquired = \varnothing$

<u>*Law . 1*</u>

$a \subseteq a$

<u>*Law . 2*</u>

$\mathsf{bsearch}\ (\ a\ ,\ b\ ,\ c\ )$
$\Rightarrow$
$a \in \{\ b\ \}$

<u>*Law . 3*</u>

$bident()\ (\ b\ ) \wedge a \subseteq c \wedge b \in d$
$\Rightarrow$
$a \times \{\ b\ \} \in c \nrightarrow d$

<u>*Law . 4*</u>

$\varnothing \in a \rightarrowtail\!\!\!\!\!\!\nrightarrow b$

<u>*Law . 5*</u>

$n \in 0\ ..\ 2147483647$
$\Rightarrow$
$n \in \mathbb{N}$

<u>*Law . 6*</u>

$\varnothing \subseteq a$

<u>*Law . 7*</u>

$a \subseteq b$

$\Rightarrow$

$a \in \mathbb{P} (b)$

<u>*Law . 8*</u>

$a \cap b \subseteq b$

<u>*Law . 9*</u>

$a = \varnothing \lor a \subseteq \varnothing$

$\Rightarrow$

$a = \varnothing$

<u>*Law . 10*</u>

$n \in \mathbb{N} \land p \in \mathbb{N}$

$\Rightarrow$

$n + p \in \mathbb{N}$

<u>*Law . 11*</u>

$a \in \{ b , a \}$

<u>*Law . 12*</u>

$f \in S \nrightarrow T \land i \in S \land v \in T$

$\Rightarrow$

$f \lhd\!\!- \{ i \mapsto v \} \in S \nrightarrow T$

<u>*Law . 13*</u>

$p = q$

$\Rightarrow$

$p - q \leq \mathit{0}$

<u>*Law . 14*</u>

$a \in \mathit{0} \,..\, \mathit{2147483647} \land b \in \mathbb{N} \land x < b \land a - \mathit{1} \leq x$

$\Rightarrow$

$a \leq b$

<u>*Law . 15*</u>

$n \in \mathbb{N} \land p \in \mathbb{N} \land p \leq n$

$\Rightarrow$

$n - p \in \mathbb{N}$

<u>*Law . 16*</u>

$a \cap b = \varnothing$

$\Rightarrow$

$a \cap b - c = \varnothing$

<u>*Law . 17*</u>

$\exists\, a . (\mathsf{card} (a) \in \mathbb{N}_1)$

D.4 Example Proof Discharged Using InterProver

Some proof obligations could not be automatically discharged. In such cases user theories were provided through the InterProver to aid in the discharge of the proof obligations. Below we show one such proof that makes use of User Theory.

Proofs (Level 6) for HotelD.mch

R2 . 4

1	$cst\ (\ HotelD\ )$	HYP
2	$ctx\ (\ HotelD\ )$	HYP
3	$inv\ (\ HotelD\ )$	HYP
4	$asn\ (\ HotelD\ )$	HYP
5	$\mathsf{card}\ (\ STATUS\ ) = 2$	2,Props.5
6	$STATUS = \{\ Available\ ,\ Taken\ \}$	2,Props.4
7	$\mathsf{ran}\ (\ RM\ ) \cap acquired = \varnothing$	3,HypExp.8
8	$\mathsf{dom}\ (\ ST \rhd \{\ Taken\ \}\ ) = \mathsf{ran}\ (\ RM\ ) \cup acquired$	3,HypExp.7
9	$ST \in ROOM \nrightarrow STATUS$	3,HypExp.1
10	$pre\ (\ R2\ )$	HYP
11	$ST\ (\ room\_inp\ ) = Available$	10,HypExp.12
12	$\neg\ (\ Available = Taken\ )$	6,5,Law.1
13	$\neg\ (\ ST\ (\ room\_inp\ ) = Taken\ )$	12,11
14	$room\_inp \notin \mathsf{ran}\ (\ RM\ )$	8,9,13,UsersTheory.2
15	$\mathsf{ran}\ (\ RM\ ) \cap (\ acquired \cup \{\ room\_inp\ \}\ ) = \varnothing$	14,7,UsersTheory.1
16	QED	DED

Laws (Level 6) for HotelD.mch

Props . 5

$ctx\ (\ HotelD\ )$

$\Rightarrow$

$\mathsf{card}\ (\ STATUS\ ) = 2$

Props . 4

$ctx\ (\ HotelD\ )$

$\Rightarrow$

$STATUS = \{\ Available\ ,\ Taken\ \}$

HypExp . 8

$inv\ (\ HotelD\ )$

$\Rightarrow$

$\mathsf{ran}\ (\ RM\ ) \cap acquired = \varnothing$

HypExp . 7

$inv\ (\ HotelD\ )$

$\Rightarrow$

$\mathsf{dom}\ (\ ST\ \rhd\ \{\ Taken\ \}\ )=\mathsf{ran}\ (\ RM\ )\cup acquired$

$\underline{HypExp\ .\ 1}$

$inv\ (\ HotelD\ )$

$\Rightarrow$

$ST\in ROOM\ \nrightarrow\ STATUS$

$\underline{HypExp\ .\ 12}$

$pre\ (\ R2\ )$

$\Rightarrow$

$ST\ (\ room\_inp\ )=Available$

$\underline{UsersTheory\ .\ 2}$

$\mathsf{dom}\ (\ c\ \rhd\ \{\ e\ \}\ )=\mathsf{ran}\ (\ a\ )\cup h\wedge c\in x\nrightarrow y\wedge\neg\ (\ c\ (\ b\ )=e\ )$

$\Rightarrow$

$b\notin\mathsf{ran}\ (\ a\ )$

$\underline{UsersTheory\ .\ 1}$

$b\notin\mathsf{ran}\ (\ a\ )\wedge\mathsf{ran}\ (\ a\ )\cap h=\varnothing$

$\Rightarrow$

$\mathsf{ran}\ (\ a\ )\cap(\ h\cup\{\ b\ \}\ )=\varnothing$

$\underline{Law\ .\ 1}$

$STATUS=\{\ Available\ ,\ Taken\ \}\wedge\mathsf{card}\ (\ STATUS\ )=2$

$\Rightarrow$

$\neg\ (\ Available=Taken\ )$

Appendix E

Example SMV specification

E.1 Input to the SMV Model Checker

The hotel example presented in Chapter 7 was executed by the SMV model checker. An example input presented to the model checker is given below. The input includes the model as well as the properties being verified. This model assumed that at most three *Reserve* transactions will be executed (given by the value of the constant *maxR* in the SMV input shown below). The SMV input for the other two models were similar to this one, except that the value of *maxR* were 1 and 2 respectively.

```
MODULE main
VAR
    input : {R1, R2, R3, CompR1, CompR2, CancelD, ReportD};
    ST : array 1..3 of boolean;
    RMI : array 1..3 of {a, b, c, d, n};
    res : 0..3;
    underway : 0..3;
    acquired : array 1..3 of boolean;
    r_inp : {1, 2, 3};
    pr_inp : {1, 2, 3};
    g_inp : {a, b, c, d};
    ReportJustExecuted : {Y, N};
    r1 : 0..3;
    r2 : 0..3;
    r3 : 0..3;
    compr1 : 0..3;
    compr2 : 0..3;
```

$--$ *Generalized Integrity Constraints*

$SPEC\ AG(RMI[1] = a -> !(RMI[2] = a)\ \&\ !(RMI[3] = a))$
$SPEC\ AG(RMI[1] = b -> !(RMI[2] = b)\ \&\ !(RMI[3] = b))$
$SPEC\ AG(RMI[1] = c -> !(RMI[2] = c)\ \&\ !(RMI[3] = c))$
$SPEC\ AG(RMI[1] = d -> !(RMI[2] = d)\ \&\ !(RMI[3] = d))$
$SPEC\ AG(RMI[2] = a -> !(RMI[1] = a)\ \&\ !(RMI[3] = a))$
$SPEC\ AG(RMI[2] = b -> !(RMI[1] = b)\ \&\ !(RMI[3] = b))$
$SPEC\ AG(RMI[2] = c -> !(RMI[1] = c)\ \&\ !(RMI[3] = c))$
$SPEC\ AG(RMI[2] = d -> !(RMI[1] = d)\ \&\ !(RMI[3] = d))$
$SPEC\ AG(RMI[3] = a -> !(RMI[1] = a)\ \&\ !(RMI[2] = a))$
$SPEC\ AG(RMI[3] = b -> !(RMI[1] = b)\ \&\ !(RMI[2] = b))$
$SPEC\ AG(RMI[3] = c -> !(RMI[1] = c)\ \&\ !(RMI[2] = c))$
$SPEC\ AG(RMI[3] = d -> !(RMI[1] = d)\ \&\ !(RMI[2] = d))$

$SPEC\ AG(res = underway + (!(RMI[1] = n)) + (!(RMI[2] = n)) + (!(RMI[3] = n)))$

$SPEC\ AG(ST[1] <-> (!(RMI[1] = n))\ |\ acquired[1])$
$SPEC\ AG(ST[2] <-> (!(RMI[2] = n))\ |\ acquired[2])$
$SPEC\ AG(ST[3] <-> (!(RMI[3] = n))\ |\ acquired[3])$

$SPEC\ AG(acquired[1] -> (RMI[1] = n))$
$SPEC\ AG(acquired[2] -> (RMI[2] = n))$
$SPEC\ AG(acquired[3] -> (RMI[3] = n))$

$--$ *Consistent Execution Property*

$SPEC\ AG((r1 = r3 + compr1) ->$
$\quad res = (!(RMI[1] = n)) + (!(RMI[2] = n)) + (!(RMI[3] = n)))$
$SPEC\ AG((r1 = r3 + compr1)\ \&\ (r2 = r3 + compr2) ->$
$\quad (ST[1] <-> !(RMI[1] = n))\ \&\ (ST[2] <-> !(RMI[2] = n))\ \&$
$\quad (ST[3] <-> !(RMI[3] = n)))$

$--$ *Sensitive Transaction Isolation Property*

$SPEC\ AG((acquired[1]\ |\ acquired[2]\ |\ acquired[3]) -> (ReportJustExecuted = N))$

$--$ *Semantic Atomicity Property*

$SPEC\ AG(!((underway > 0)\ \&\ (r2 < maxR)\ \&(((r\_inp = 1)\ \&\ !ST[1])\ |$
$\quad ((r\_inp = 2)\ \&\ !ST[2])\ |\ ((r\_inp = 3)\ \&!ST[3])))\&!((underway > 0)\ \&$
$\quad (r3 < maxR)\ \&!(RMI[1] = g\_inp)\ \&\ !(RMI[2] = g\_inp)\ \&\ !(RMI[3] = g\_inp)$
$\quad \&(((pr\_inp = 1)\ \&\ acquired[1])\ |\ ((pr\_inp = 2)\ \&\ acquired[2])\ |((pr\_inp = 3)\ \&$
$\quad acquired[3])))\&!(res > 0\ \&\ underway > 0\ \&\ (compr1 < maxR))\ \&!((((pr\_inp = 1)$
$\quad \&\ acquired[1])\ |\ ((pr\_inp = 2)\ \&\ acquired[2])\ |((pr\_inp = 3)\ \&\ acquired[3]))$
$\quad \&\ (compr2 < maxR))-> (r1 = r3 + compr1)\ \&\ (r2 = r3 + compr2))$

```
DEFINE
    total := 3;
    maxR := 1;

ASSIGN
    init(res) := 0;
    next(res) := case
        (input = R1) & (res < total) & (underway < total)
            & (r1 < maxR) : res + 1;
        (input = CompR1) & (res > 0) & (underway > 0) &
            (compr1 < maxR) : res - 1;
        (input = CancelD) & (g_inp = a) &(res > 0) &
            (RMI[1] = a | RMI[2] = a | RMI[3] = a) : res - 1;
        (input = CancelD) & (g_inp = b) & (res > 0) &
            (RMI[1] = b | RMI[2] = b | RMI[3] = b) : res - 1;
        (input = CancelD) & (g_inp = c) & (res > 0)&
            (RMI[1] = c | RMI[2] = c | RMI[3] = c) : res - 1;
        (input = CancelD) & (g_inp = d) & (res > 0) &
            (RMI[1] = d | RMI[2] = d | RMI[3] = d) : res - 1;
        1 : res;
    esac;

    init(underway) := 0;
    next(underway) := case
        (input = R1) & (res < total) & (underway < total)
            & (r1 < maxR) : underway + 1;
        (input = R3) & (underway > 0) & (pr_inp = 1) & acquired[1] &
            !(RMI[1] = g_inp) & !(RMI[2] = g_inp)&!(RMI[3] = g_inp)&
            (r3 < maxR) : underway - 1;
        (input = R3) & (underway > 0) & (pr_inp = 2) & acquired[2]&
            !(RMI[1] = g_inp) & !(RMI[2] = g_inp)&!(RMI[3] = g_inp)&
            (r3 < maxR) : underway - 1;
        (input = R3) & (underway > 0) & (pr_inp = 3) & acquired[3]&
            !(RMI[1] = g_inp) & !(RMI[2] = g_inp) & !(RMI[3] = g_inp)&
            (r3 < maxR) : underway - 1;
        (input = CompR1) & (res > 0) & (underway > 0)
            & (compr1 < maxR) : underway - 1;
        1 : underway;
    esac;
```

$init(ST[1]) \ := 0;$
$next(ST[1]) \ := case$
$\quad (input \ = \ R2) \ \& \ (underway > 0) \ \& \ (r\_inp \ = \ 1) \ \& \ !(ST[1])$
$\quad\quad \& \ (r2 < maxR) : 1;$
$\quad (input \ = \ CompR2) \ \& \ (pr\_inp \ = \ 1) \ \& \ acquired[1] \ \& \ (compr2 < maxR) : 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ a) \ \& \ (res > 0) \ \& \ (RMI[1] \ = \ a) : 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ b) \ \& \ (res > 0) \ \& \ (RMI[1] \ = \ b) : 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ c) \ \& \ (res > 0) \ \& \ (RMI[1] \ = \ c) : 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ d) \ \& \ (res > 0) \ \& \ (RMI[1] \ = \ d) : 0;$
$\quad 1 : ST[1];$
$esac;$

$init(ST[2]) := 0;$
$next(ST[2]) := case$
$\quad (input \ = \ R2) \ \& \ (underway > 0)\&(r\_inp \ = \ 2) \ \& \ !(ST[2])$
$\quad\quad \& \ (r2 < maxR) : 1;$
$\quad (input \ = \ CompR2) \ \& \ (pr\_inp \ = \ 2) \ \& \ acquired[2] \ \& \ (compr2 < maxR) : 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ a) \ \& \ (res > 0) \ \& \ (RMI[2] \ = \ a) : 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ b) \ \& \ (res > 0) \ \& \ (RMI[2] \ = \ b) : 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ c) \ \& \ (res > 0) \ \& \ (RMI[2] \ = \ c) : 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ d)\&(res > 0) \ \& \ (RMI[2] \ = \ d) : 0;$
$\quad 1 \ : \ ST[2];$
$esac;$

$init(ST[3]) \ := \ 0;$
$next(ST[3]) \ := \ case$
$\quad (input \ = \ R2) \ \& \ (underway > 0) \ \& \ (r\_inp \ = \ 3) \ \& \ !(ST[3])$
$\quad\quad \& \ (r2 < maxR) \ : \ 1;$
$\quad (input \ = \ CompR2) \ \& \ (pr\_inp \ = \ 3) \ \& \ acquired[3] \ \& \ (compr2 < maxR) \ : \ 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ a) \ \& \ (res > 0) \ \& \ (RMI[3] \ = \ a) \ : \ 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ b) \ \& \ (res > 0) \ \& \ (RMI[3] \ = \ b) \ : \ 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ c) \ \& \ (res > 0) \ \& \ (RMI[3] \ = \ c) \ : \ 0;$
$\quad (input \ = \ CancelD) \ \& \ (g\_inp \ = \ d) \ \& \ (res > 0) \ \& \ (RMI[3] \ = \ d) \ : \ 0;$
$\quad 1 \ : \ ST[3];$
$esac;$

$init(acquired[1])\ :=\ 0;$
$next(acquired[1])\ :=\ case$
$(input\ =\ R2)\ \&\ (r\_inp\ =\ 1)\ \&\ (underway > 0)\ \&$
$!(ST[1])\ \&\ (r2 < maxR)\ :\ 1;$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 1)\ \&\ acquired[1]\ \&$
$!(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)$
$\&\ !(RMI[3]\ =\ g\_inp)\ \&\ (r3 < maxR)\ :\ 0;$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 1)\ \&\ acquired[1]\ \&\ (compr2 < maxR)\ :\ 0;$
$1\ :\ acquired[1];$
$esac;$

$init(acquired[2])\ :=\ 0;$
$next(acquired[2])\ :=\ case$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 2)$
$\&\ !(ST[2])\ \&\ (r2 < maxR)\ :\ 1;$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 2)$
$\&\ acquired[2]\ \&\ !(RMI[1]\ =\ g\_inp)$
$\&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)\ \&\ (r3 < maxR)\ :\ 0;$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 2)\ \&\ acquired[2]\ \&\ (compr2 < maxR)\ :\ 0;$
$1\ :\ acquired[2];$
$esac;$

$init(acquired[3])\ :=\ 0;$
$next(acquired[3])\ :=\ case$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 3)$
$\&\ !(ST[3])\ \&\ (r2 < maxR)\ :\ 1;$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 3)$
$\&\ acquired[3]\ \&\ !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)$
$\&\ !(RMI[3]\ =\ g\_inp)\ \&\ (r3 < maxR)\ :\ 0;$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 3)\ \&\ acquired[3]\ \&\ (compr2 < maxR)\ :\ 0;$
$1\ :\ acquired[3];$
$esac;$

$init(RMI[1])\ :=\ n;$
$next(RMI[1])\ :=\ case$
$\quad (input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 1)\ \&\ acquired[1]$
$\qquad \&\ !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)$
$\qquad (r3 < maxR)\ :\ g\_inp;$
$\quad (input\ =\ CancelD)\ \&\ (RMI[1]\ =\ g\_inp)\ :\ n;$
$\quad 1\ :\ RMI[1];$
$esac;$

$init(RMI[2])\ :=\ n;$
$next(RMI[2])\ :=\ case$
$\quad (input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 2)\ \&\ acquired[2]\ \&$
$\qquad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)$
$\qquad \&\ (r3 < maxR)\ :\ g\_inp;$
$\quad (input\ =\ CancelD)\ \&\ (RMI[2]\ =\ g\_inp)\ :\ n;$
$\quad 1\ :\ RMI[2];$
$esac;$

$init(RMI[3])\ :=\ n;$
$next(RMI[3])\ :=\ case$
$\quad (input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 3)\ \&\ acquired[3]\ \&$
$\qquad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)$
$\qquad \&\ (r3 < maxR)\ :\ g\_inp;$
$\quad (input\ =\ CancelD)\ \&\ (RMI[3]\ =\ g\_inp)\ :\ n;$
$\quad 1\ :\ RMI[3];$
$esac;$

$init(ReportJustExecuted)\ :=\ N;$
$next(ReportJustExecuted)\ :=\ case$
$\quad (input\ =\ ReportD)\ \&\ !(acquired[1])\ \&\ !(acquired[2])\ \&\ !(acquired[3])\ :\ Y;$
$\quad 1\ :\ N;$
$esac;$

$init(r1)\ :=\ 0;$
$next(r1)\ :=\ case$
$\quad (input\ =\ R1)\ \&\ (res < total)\ \&\ (underway < total)$
$\qquad \&\ (r1 < maxR)\ :\ r1 + 1;$
$\quad 1\ :\ r1;$
$esac;$

$init(r2) \ := \ 0;$
$next(r2) \ := \ case$
$\quad (input \ = \ R2) \ \& \ (underway > 0) \ \& \ (r\_inp \ = \ 1)$
$\qquad \& \ !(ST[1]) \ \& \ (r2 < maxR) \ : \ r2 + 1;$
$\quad (input \ = \ R2) \ \& \ (underway > 0) \ \& \ (r\_inp \ = \ 2)$
$\qquad \& \ !(ST[2]) \ \& \ (r2 < maxR) \ : \ r2 + 1;$
$\quad (input \ = \ R2) \ \& \ (underway > 0) \ \& \ (r\_inp \ = \ 3)$
$\qquad \& \ !(ST[3]) \ \& \ (r2 < maxR) \ : \ r2 + 1;$
$\quad 1 \ : \ r2;$
$esac;$

$init(r3) \ := \ 0;$
$next(r3) \ := \ case$
$\quad (input \ = \ R3) \ \& \ (underway > 0) \ \& \ (pr\_inp \ = \ 1) \ \& \ acquired[1] \ \&$
$\qquad !(RMI[1] \ = \ g\_inp) \ \& \ !(RMI[2] \ = \ g\_inp) \ \& \ !(RMI[3] \ = \ g\_inp) \ \&$
$\qquad (r3 < maxR) \ : \ r3 + 1;$
$\quad (input \ = \ R3) \ \& \ (underway > 0) \ \& \ (pr\_inp \ = \ 2) \ \& \ acquired[2] \ \&$
$\qquad !(RMI[1] \ = \ g\_inp) \ \& \ !(RMI[2] \ = \ g\_inp) \ \& \ !(RMI[3] \ = \ g\_inp) \ \&$
$\qquad (r3 < maxR) \ : \ r3 + 1;$
$\quad (input \ = \ R3) \ \& \ (underway > 0) \ \& \ (pr\_inp \ = \ 3) \ \& \ acquired[3] \ \&$
$\qquad !(RMI[1] \ = \ g\_inp) \ \& \ !(RMI[2] \ = \ g\_inp) \ \& \ !(RMI[3] \ = \ g\_inp) \ \&$
$\qquad (r3 < maxR) \ : \ r3 + 1;$
$\quad 1 \ : \ r3;$
$esac;$

$init(compr1) \ := \ 0;$
$next(compr1) \ := \ case$
$\quad (input \ = \ CompR1) \ \& \ (res > 0) \ \& \ (underway > 0) \ \& \ (compr1 < maxR)$
$\qquad : \ compr1 + 1;$
$\quad 1 \ : \ compr1;$
$esac;$

$init(compr2) \ := \ 0;$
$next(compr2) \ := \ case$
$\quad (input \ = \ CompR2) \ \& \ (pr\_inp \ = \ 1) \ \& \ acquired[1] \ \& \ (compr2 < maxR)$
$\qquad : \ compr2 + 1;$
$\quad (input \ = \ CompR2) \ \& \ (pr\_inp \ = \ 2) \ \& \ acquired[2] \ \& \ (compr2 < maxR)$
$\qquad : \ compr2 + 1;$
$\quad (input \ = \ CompR2) \ \& \ (pr\_inp \ = \ 3) \ \& \ acquired[3] \ \& \ (compr2 < maxR)$
$\qquad : \ compr2 + 1;$
$\quad 1 \ : \ compr2;$
$esac;$

$next(g\_inp)\ :=\ case$
$(input\ =\ R1)\ :\ \{a, b, c, d\};$
$(input\ =\ CancelD)\ :\ \{a, b, c, d\};$
$1\ :\ g\_inp;$
$esac;$

$next(r\_inp)\ :=\ case$
$(input\ =\ R1)\ :\ \{1, 2, 3\};$
$1\ :\ r\_inp;$
$esac;$

$next(pr\_inp)\ :=\ case$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 1)\ \&\ !(ST[1])\ \&\ (r2 < maxR)$
$\&\ acquired[2]\ \&\ acquired[3]\ :\ \{1, 2, 3\};$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 2)\ \&\ !(ST[2])\ \&\ (r2 < maxR)$
$\&\ acquired[1]\ \&\ acquired[3]\ :\ \{1, 2, 3\};$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 3)\ \&\ !(ST[3])\ \&\ (r2 < maxR)$
$\&\ acquired[1]\ \&\ acquired[2]\ :\ \{1, 2, 3\};$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 1)\ \&\ !(ST[1])\ \&\ (r2 < maxR)$
$\&\ !acquired[3]\ \&\ acquired[2]\ :\ \{1, 2\};$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 1)\ \&\ !(ST[1])\ \&\ (r2 < maxR)$
$\&\ !acquired[1]\ \&\ !acquired[2]\ :\ 1;$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 2)\ \&\ !(ST[2])\ \&\ (r2 < maxR)$
$!acquired[1]\ \&\ acquired[3]\ :\ \{2, 3\};$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 2)\ \&\ !(ST[2])\ \&\ (r2 < maxR)$
$!acquired[1]\ \&\ !acquired[2]\ :\ 2;$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 3)\ \&\ !(ST[3])\ \&\ (r2 < maxR)$
$\&\ !acquired[1]\ \&\ acquired[2]\ :\ \{2, 3\};$
$(input\ =\ R2)\ \&\ (underway > 0)\ \&\ (r\_inp\ =\ 3)\ \&\ !(ST[3])\ \&\ (r2 < maxR)$
$\&\ !acquired[1]\ \&\ !acquired[2]\ :\ 3;$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 1)\ \&\ acquired[1]\ \&\ (compr2 < maxR)$
$\&\ acquired[2]\ \&\ acquired[3]\ :\ \{2, 3\};$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 2)\ \&\ acquired[2]\ \&\ (compr2 < maxR)$
$\&\ acquired[1]\ \&\ acquired[3]\ :\ \{1, 3\};$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 3)\ \&\ acquired[3]\ \&\ (compr2 < maxR)$
$\&\ acquired[1]\ \&\ acquired[2]\ :\ \{1, 2\};$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 1)\ \&\ acquired[1]\ \&\ (compr2 < maxR)$
$\&\ acquired[2]\ \&\ !acquired[3]\ :\ 2;$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 1)\ \&\ acquired[1]\ \&\ (compr2 < maxR)$
$\&\ !acquired[2]\ \&\ acquired[3]\ :\ 3;$

$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 2)\ \&\ acquired[2]\ \&\ (compr2 < maxR)$
$\quad \&\ acquired[1]\ \&\ !acquired[3]\ :\ 1;$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 2)\ \&\ acquired[2]\ \&\ (compr2 < maxR)$
$\quad \&\ !acquired[1]\ \&\ acquired[3]\ :\ 3;$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 3)\ \&\ acquired[3]\ \&\ (compr2 < maxR)$
$\quad \&\ acquired[1]\ \&\ !acquired[2]\ :\ 1;$
$(input\ =\ CompR2)\ \&\ (pr\_inp\ =\ 3)\ \&\ acquired[3]\ \&\ (compr2 < maxR)$
$\quad \&\ !acquired[1]\ \&\ acquired[2]\ :\ 2;$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 1)\ \&\ acquired[1]\ \&$
$\quad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)\ \&$
$\quad (r3 < maxR)\ \&\ acquired[2]\ \&\ acquired[3]\ :\ \{2,3\};$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 2)\ \&\ acquired[1]\ \&$
$\quad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)\ \&$
$\quad (r3 < maxR)\ \&\ acquired[2]\ \&\ acquired[3]\ :\ \{1,3\};$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 3)\ \&\ acquired[1]\ \&$
$\quad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)\ \&$
$\quad (r3 < maxR)\ \&\ acquired[2]\ \&\ acquired[3]\ :\ \{1,2\};$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 1)\ \&\ acquired[1]\ \&$
$\quad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)\ \&$
$\quad (r3 < maxR)\ \&\ !acquired[2]\ \&\ acquired[3]\ :\ 3;$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 1)\ \&\ acquired[1]\ \&$
$\quad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)\ \&$
$\quad (r3 < maxR)\ \&\ acquired[2]\ \&\ !acquired[3]\ :\ 2;$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 2)\ \&\ !acquired[1]\ \&$
$\quad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)\ \&$
$\quad (r3 < maxR)\ \&\ acquired[2]\ \&\ acquired[3]\ :\ 3;$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 2)\ \&\ acquired[1]\ \&$
$\quad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)\ \&$
$\quad (r3 < maxR)\ \&\ acquired[2]\ \&\ !acquired[3]\ :\ 1;$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 3)\ \&\ !acquired[1]\ \&$
$\quad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)\ \&$
$\quad (r3 < maxR)\ \&\ acquired[2]\ \&\ acquired[3]\ :\ 2;$
$(input\ =\ R3)\ \&\ (underway > 0)\ \&\ (pr\_inp\ =\ 3)\ \&\ acquired[1]\ \&$
$\quad !(RMI[1]\ =\ g\_inp)\ \&\ !(RMI[2]\ =\ g\_inp)\ \&\ !(RMI[3]\ =\ g\_inp)\ \&$
$\quad (r3 < maxR)\ \&\ !acquired[2]\ \&\ acquired[3]\ :\ 1;$
$1\ :\ pr\_inp;$

$esac;$

E.2 Output produced by the SMV Model Checker

The above input file when submitted to the SMV model checker produced the following output.

$-- \ specification \ AG \ (RMI[1] \ = \ a \ -> \ !RMI[2] \ = \ a \ \& \ !RMI[3] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[1] \ = \ b \ -> \ !RMI[2] \ = \ b \ \& \ !RMI[3] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[1] \ = \ c \ -> \ !RMI[2] \ = \ c \ \& \ !RMI[3] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[1] \ = \ d \ -> \ !RMI[2] \ = \ d \ \& \ !RMI[3] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[2] \ = \ a \ -> \ !RMI[1] \ = \ a \ \& \ !RMI[3] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[2] \ = \ b \ -> \ !RMI[1] \ = \ b \ \& \ !RMI[3] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[2] \ = \ c \ -> \ !RMI[1] \ = \ c \ \& \ !RMI[3] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[2] \ = \ d \ -> \ !RMI[1] \ = \ d \ \& \ !RMI[3] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[3] \ = \ a \ -> \ !RMI[1] \ = \ a \ \& \ !RMI[2] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[3] \ = \ b \ -> \ !RMI[1] \ = \ b \ \& \ !RMI[2] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[3] \ = \ c \ -> \ !RMI[1] \ = \ c \ \& \ !RMI[2] \ ... \ is \ true$
$-- \ specification \ AG \ (RMI[3] \ = \ d \ -> \ !RMI[1] \ = \ d \ \& \ !RMI[2] \ ... \ is \ true$
$-- \ specification \ AG \ res \ = \ underway \ + \ (!RMI[1] \ = \ n) \ + \ (!RM... \ is \ true$
$-- \ specification \ AG \ (ST[1] \ <-> !RMI[1] \ = \ n \ | \ acquired[1]) \ is \ true$
$-- \ specification \ AG \ (ST[2] \ <-> !RMI[2] \ = \ n \ | \ acquired[2]) \ is \ true$
$-- \ specification \ AG \ (ST[3] \ <-> !RMI[3] \ = \ n \ | \ acquired[3]) \ is \ true$
$-- \ specification \ AG \ (acquired[1] -> \ RMI[1] \ = \ n) \ is \ true$
$-- \ specification \ AG \ (acquired[2] -> \ RMI[2] \ = \ n) \ is \ true$
$-- \ specification \ AG \ (acquired[3] -> \ RMI[3] \ = \ n) \ is \ true$
$-- \ specification \ AG \ (r1 \ = \ r3 \ + \ compr1 \ -> \ res \ = \ (!RMI[1] \ = ... \ is \ true$
$-- \ specification \ AG \ (r1 \ = \ r3 \ + \ compr1 \ \& \ r2 \ = \ r3 \ + \ compr2 \ ... \ is \ true$
$-- \ specification \ AG \ (acquired[1] \ | \ acquired[2] \ | \ acquired... \ is \ true$
$-- \ specification \ AG \ (!(underway \ > \ 0 \ \& \ r2 \ < \ maxR \ \& \ (r_inp \ ... \ is \ true$
$resources \ used:$
$user \ time: \ 15.7167 \ s, \ system \ time: \ 0.416667 \ s$
$BDD \ nodes \ allocated: \ 50251$
$Bytes \ allocated: \ 1769472$
$BDD \ nodes \ representing \ transition \ relation: \ 6519 \ + \ 1$

Curriculum Vitae

Indrakshi Ray was born on June 25, 1966, in Calcutta, India, and is a citizen of the Republic of India. She received her Bachelor of Engineering in Computer Science and Technology from Bengal Engineering College (affiliated to the Calcutta University at that time), India, in 1988. In 1991, she received her Master of Engineering in Computer Science and Engineering from Jadavpur University, India. Between 1988 and 1989 Indrakshi worked as a software engineer with ORG Systems, India. She also worked as a teaching assistant while studying at the Florida State University, Tallahassee, during the period Fall 91 to Summer 93. Indrakshi joined the doctoral program in Information Technology at the George Mason University, Fairfax in Fall 1993. During the period Fall 93 till Summer 97, she was a Graduate Research Assistant at the Center for Secure Information Systems at George Mason University.

Indrakshi is a National Scholar of India for the year 1982. She is a member of the IFIP Technical Committee on Security and Protection in Information Processing, WG 11.3 Database Security. Indrakshi will join the faculty of the Computer and Information Sciences Department at the University of Michigan, Dearborn in Fall 1997.

This dissertation was typeset with LaTeX[‡] by the author.

[‡] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.