



ELSEVIER

Information Sciences 129 (2000) 155–195

INFORMATION
SCIENCES

AN INTERNATIONAL JOURNAL

www.elsevier.com/locate/ins

Using semantic correctness in multidatabases to achieve local autonomy, distribute coordination, and maintain global integrity

Indrakshi Ray¹, Paul Ammann, Sushil Jajodia^{*,2}

ISSE Department, George Mason University, Mail Stop 4A4, Fairfax, VA 22030, USA

Abstract

A multidatabase 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. Second, management of global transactions must be distributed to avoid bottlenecks and to tolerate failure in the global database. Third, both local and global integrity constraints must be maintained. Finally, concurrent processing of transactions requires that execution histories be correct, an objective traditionally achieved with serializability. Although alternate forms of correctness have been proposed, none of the solutions advanced to date has simultaneously achieved all four requirements. We propose a transaction processing model that uses a semantics-based notion of correctness to achieve all four requirements simultaneously for applications that satisfy a certain set of properties. The support required for global transactions forms a separate layer at each site that respects both design autonomy and execution autonomy. Global transactions are managed at each site via a successor set mechanism so that site and communication failures do not impede transactions at operating sites. Semantic correctness encompasses three properties: ensuring that local and global integrity constraints are maintained, that transactions output consistent data and that all partially executed transactions can complete. A fourth property ensures that the successor set description is a valid refinement of the

* Corresponding author. Tel.: +1-703-993-1653; fax: +1-703-993-1638.

E-mail address: jajodia@gmu.edu (S. Jajodia).

¹ CIS Department, University of Michigan-Dearborn, 4901 Evergreen Road, Dearborn, MI 48128, USA.

² Partially supported by National Science Foundation under grant IRI-9633541 and by National Security Agency under grants MDA904-96-1-0103 and MDA904-96-1-0104.

specification. These four properties must be proved for applications executed by our model. We show how model checking can automate, in part, the verification of the properties. © 2000 Elsevier Science Inc. All rights reserved.

Keywords: Database; Disturbed database; Multidatabase; Semantic correctness; Model checking; Transaction processing

1. Introduction

The multidatabase area has received considerable attention [3,4,6,7,10, 11,13,18,19,24]. The goal of a multidatabase is to integrate a heterogeneous collection of existing databases, thereby enabling *global* transactions to span the multiple *local* databases. Four typically contradictory goals drive the integration: respect for local autonomy, distribution of global transaction management, maintenance of local and global integrity constraints, and correctness of execution histories. No existing multidatabase proposal satisfies all of these goals; but the multidatabase model proposed in this paper does for applications that satisfy a certain set of properties. Our notion of semantic correctness allows us to achieve local autonomy of local databases, distribute coordination of global transactions, and maintain local and global integrity constraints. Before we outline our approach, we elaborate the four multidatabase goals.

Local autonomy. The local databases typically exist long before being organized as a multidatabase. Furthermore the local databases may be owned by separate, and possibly competing, organizations. Thus each local database requires both design autonomy to accommodate its diverse legacy nature and execution autonomy to minimize the outside interference of global transactions with local processing [10,20].

Standard protocols often violate local autonomy. For example, the two phase commit protocol, which is often 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 subtransactions at their discretion, the two-phase commit protocol also violates execution autonomy.

Local autonomy forces the local subtransactions of global transactions to be treated as ordinary local transactions. For example, the local database is free to choose any concurrency control algorithm, and global transactions must accommodate that choice. Mechanisms for managing global transactions must be built on top of – as opposed to inside – the local database.

Distributed coordination. The local databases in a multidatabase may fail, or be disconnected from the network, or simply choose to limit their degree of global cooperation at any time. To provide continued service to subsets of a

multidatabase, global transaction processing must be tolerant of single points of failure. For this reason, global transaction processing should be distributed instead of centralized. As we elaborate in the related work section, most multidatabase proposals do not satisfy this criteria.

Global integrity. Local databases are charged with maintaining local integrity constraints. Global integrity constraints are more difficult to maintain, particularly in the absence of a centralized global transaction manager. Some researchers have argued against having global integrity constraints in multidatabases [13], but we take the position in this paper that global constraints are a natural part of some applications, and that means for implementing such constraints should be provided.

Correctness. Syntactic correctness criteria use some notion of serializability. Serializability obligates the mechanism processing global transactions to make consistent ordering decisions at each local database, which is problematic for multidatabases. Most works using the standard serializability notion [5] for multidatabases [7,18] require a centralized global transaction manager with a relatively high overhead. A decentralized approach that guarantees serializable execution of multidatabase transactions, but requires the local databases to support a visible prepared-to-commit state, was proposed by Batra et al. [6]. Using weaker serializability notions for multidatabases [10,13,19] impose severe constraints on which applications can be accommodated. Some multidatabase models [20] require a stronger notion of correctness than serializability due to the possibility that a global transaction will generate output via some local subtransaction and then semantically abort by executing a compensating step. See Section 2 for a detailed review. Semantic notions of correctness have also been applied to multidatabases [24], but without the simultaneous achievement of local autonomy, distributed coordination, and global integrity.

In this paper, we do not use serializability as the correctness criterion, but introduce a semantic correctness model tailored to multidatabases. The result is a property-oriented approach to analyzing applications. The properties are: (1) semantic atomicity: a global transaction commits or compensates for all of its subtransactions, (2) consistent execution: global integrity constraints are restored upon completion of partially executed global transactions, and (3) sensitive transaction isolation: outputs always appear to have been generated from a consistent state. Together, these three properties encompass two of the goals, namely, maintenance of global integrity constraints and correctness. Note that these three new properties replace their syntactic counterparts, namely atomicity, consistency and isolation, which are used in the standard transaction processing model.

We introduce a successor set mechanism for managing global transactions. A fourth property, the (4) valid successor set property, ensures that a successor set description is a valid refinement of the global transactions. We describe the integration of successor set descriptions for global transactions with local

transaction processing. The successor set mechanism yields the remaining two goals, namely, local autonomy and distributed coordination of global transactions. Thus, for applications that satisfy the necessary four properties, our approach provides for all four of the desired goals for multidatabases.

There are two possible outcomes to analyzing an application. One outcome is that the application does indeed enjoy the four properties. The other outcome is that the application does not satisfy one or more of the four properties, in which case the application developer must either revise the application or select an alternate approach. In this paper we give assurance whether the properties indeed hold for a given application; if the application does not possess the necessary properties we do not suggest an approach of modifying the application to ensure the satisfaction of the properties. Appendix A illustrates the formalization of an example in the specification language *Object Z*. The *Object Z* language [9] was chosen because it allows predicates on histories to be specified as temporal logic formulae. The appendices contain arguments that the four properties hold for the example.

An important issue is how well our model scales up to real world applications. The necessary properties must be demonstrated for applications which must be implemented by our model. For the purpose of this paper we use the *Object Z* specification language and analyze the specifications by hand. However, for real world applications this may not be feasible; for such applications it is necessary to automate to the extent possible the process of discharging the proof obligations. Therefore, we also examine model checking as an automated approach to verifying the properties. Model checking requires the system to be verified be represented as a finite state machine. Since software systems are in general infinite state machines, finite state abstractions of the software systems must be developed which can then be analyzed by the model checker [26]. We show how the running example can be abstracted into a finite state machine and the properties verified in the abstraction by the SMV model checker [17]. The verification in the abstraction yields informal confidence that the properties hold in the original.

The rest of the paper is organized as follows. Section 2 discusses the related work done in this area. In Sections 3 and 4 we present our model. Section 5 outlines an efficient mechanism for ensuring semantic correctness. Section 6 discusses the extensions required to support global transactions. Section 7 describes how a model checker can be used to verify semantic correctness for an example application. Finally Section 8 concludes the paper.

2. Related work

Several schemes [7,18] 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.

A decentralized mechanism to ensure global serializability has been proposed by Batra et al. [6]. Each site consists of a global transaction manager (GTM) and a set of servers. The site at which the global transaction is submitted becomes the coordinator of that transaction. The coordinator assigns a unique timestamp to the global transaction, decomposes the transaction into subtransactions, and forwards the subtransactions to the participating sites. The GTM at the participating site allocates a server for the subtransaction which monitors its execution and interacts with the GTM.

Each local database stores a data item called a ticket or global timestamp (GTS). Each subtransaction executing at the local database is required to read and write this ticket. The ticket operations introduce local conflicts on the subtransactions executing at a site and thereby impose a serialization order on the subtransactions. When a global transaction commits, the ticket or GTS is updated to the timestamp of the committed transaction. A subtransaction whose timestamp is less than GTS is aborted.

When a subtransaction completes execution, the server informs the GTM of the outcome and enters a visible prepared-to-commit state (a state in which the subtransaction will not be aborted unilaterally by the local database). The GTM then forwards this message to the coordinator GTM which then sends a commit or abort message to the participating GTMs. In response to this, the participating GTMs either commit or abort the subtransactions. When all the subtransactions have committed or aborted, the transaction is said to be completed. Note that we do not use serializability as the correctness criterion and consequently do not require the local databases to provide a visible prepared-to-commit state.

Mehrotra et al. [20] present the *GTM commit protocol* for ensuring semantic atomicity [12]. 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, as compensatable, pivot, and retrievable. Compensatable subtransactions can be compensated by executing compensating subtransactions, retrievable subtransactions can be successfully executed if retried a sufficient number of times, and pivot subtransactions can be neither compensated nor retried. Although the authors [20] do not formally specify how to classify subtransactions, a separate work [21] 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 retrievable, otherwise not. Similarly if the compensating code does not contain preconditions involving database variables then the subtransaction is compensatable. For example, Withdraw cannot, in general, compensate for

Deposit because Withdraw has a precondition, namely that there are sufficient funds. However, Deposit, which has no such preconditions, can compensate for Withdraw. The GTM commit protocol requires the subtransactions of a global transaction to be committed in the following order: compensatable, pivot and retrievable. Note that each global transaction can have at most one pivot transaction. 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. The other sites on receiving this message either execute compensating subtransactions or abort the subtransactions depending on whether or not the subtransactions have committed. If the local DBMS aborts a retrievable subtransaction, then this subtransaction is retried until it executes successfully. After completing a subtransaction, the local DBMS must wait for a commit message from the GTM before committing the subtransaction.

There are several differences between our approach and the one presented by Mehrotra et al. [20]. The major difference is that we do not require a centralized 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 Section 3.4). In our model, the local databases have complete autonomy to abort or commit the subtransactions executing at the site. This contrasts the approach presented by Mehrotra et al., 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 their approach, 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 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 [10,13,24,19] maintain that traditional notion of serializability is needlessly restrictive in a multidatabase environment and have proposed alternative correctness criteria. Du and Elmagarmid [10] 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. [19] 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. [16] 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 et al. [24] 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 they propose require a centralized global transaction manager for coordinating global transactions. These algorithms are based on graphs and have significant run-time overhead.

Garcia–Molina advocates the use of the Saga model for handling multidatabase applications. The Saga model [14] 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. For instance, Example 5 in Section 4 cannot be handled by the Saga model.

Although a lot of work has been done in multidatabase concurrency control, much less literature appear in multidatabase recovery. Towards this end, Breitbart et al. [4] argue the need for local rigorous schedulers in a multidatabase environment. Multidatabase recovery has been discussed at length by Georgakopoulos [11] where the author proposes a multidatabase recoverability requirement and describes a recovery mechanism that satisfies this requirement.

3. The model

A multidatabase consists of a collection of local databases. At any given time, the *state* of a local database is determined by the values of the objects in the database. A change in the value of a database object changes the state. The global state comprises the states of all the individual local databases. *Integrity constraints* are predicates defined over the objects. Integrity constraints 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. Note that, in general, different local databases will have different local integrity constraints. Our model assumes that global integrity constraints do not violate any local integrity constraints. The local state is said to be *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.

A multidatabase application consists of a set of transactions. A *transaction* is an operation that transforms one 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. Together, preconditions and postconditions must ensure that if a transaction executes on a consistent state, the result is again a consistent state. A transaction is classified as local or global depending on whether the transaction accesses objects from one or more than one database.

Example 1 (*Basic banking enterprise example*). We consider a simple banking example adapted from the work of Mehrotra et al. [20]. 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 banking enterprise has two local integrity constraints: the balance of each

account is positive, and an account is associated with only one balance.³ At present, the application has no global integrity constraints; these are introduced in Section 4.

The banking enterprise has a number of local and global transactions. The local transactions are *Ldeposit*, *Lwithdraw*, *Ltransfer*, and *Laudit*. There are two global transactions, *Gtransfer* and *Gaudit*. The precondition of *Ldeposit* is that the account must exist in the branch. The postcondition of *Ldeposit* adds the amount deposited to the balance. *Lwithdraw* and *Ltransfer* are similarly defined. *Laudit* has no preconditions and produces as output the account balances of all the accounts in the branch. 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 paper we consider an instance of the banking enterprise with one branch (Branch *B*) and one head office (Head Office *H*). 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. Where it is not necessary to distinguish the operations of the head office from that of the branch, we omit the prefixes (*B.* and *H.*) when referring to the operations (for example, just use *Ltransfer*).

3.1. Decomposition of global transactions

To satisfy the requirements for multidatabases outlined in Section 1, it is necessary to execute a global transaction as a collection of local subtransactions rather than as an atomic unit. We assume that at one most subtransaction from a given global transaction executes at each site. A *decomposition* of a global transaction is a partial order of subtransactions. A global transaction T_i is decomposed into $S_{i1}, S_{i2}, \dots, S_{in}$, where S_{ij} denotes the subtransaction of T_i that executes at site j .

Example 2. In our banking example, the *Gtransfer* subtransaction is decomposed into two atomic subtransactions *Gtransfer1* and *Gtransfer2*, where *Gtransfer1*, *Gtransfer2* are specified in the same way as *Lwithdraw* and

³ Although we can model the case where different local databases have different integrity constraints (for example, one branch may not allow negative account balances whereas another one may allow it), to keep the example simple we assume that all branches have the same local integrity constraints.

Ldeposit. The *Gaudit* transaction is decomposed into two subtransactions *Gaudit1* and *Gaudit2*, where *Gaudit1* is similar to an *Laudit* at *H* and *Gaudit2* is similar to an *Laudit* at *B*.

3.2. Compensating subtransactions

In our model, a global transaction may abort after committing one or more subtransactions. The committed subtransactions cannot be undone by physically restoring the state that existed prior to the execution of the subtransaction. This is because other subtransactions or local transactions may have been exposed to the updates of the committed subtransaction. We address this problem with compensating subtransactions [12]. (Note that, committed subtransactions that must be compensated will not cause global consistency problems, due to the sensitive transaction isolation property introduced later on.) As pointed out by Mehrotra et al. [20], some subtransactions are not compensatable and some subtransactions do not require compensation. For each compensatable subtransaction S_{ij} the application developer may specify a compensating subtransaction C_{ij} which semantically undoes the actions of S_{ij} . A compensating subtransaction is never needed for the last subtransaction, if any, of the global transaction. In cases where multiple subtransactions can execute concurrently and there is no fixed last subtransaction, compensating subtransactions must be specified for all the compensatable subtransactions. We assume that C_{ij} runs at the same site as S_{ij} . A compensating subtransaction C_{ij} is like any another subtransaction of T_i except that it is invoked only when T_i aborts.

Example 3. The compensating subtransaction for *Gtransfer1* is *Gtrans1comp*. The execution of *Gtrans1comp* is similar to performing an *LDeposit* to the account from which the withdrawal was made in *Gtransfer1*. We treat the subtransaction *Gaudit1* as uncompensatable because it generates an output.

Where it is not necessary to distinguish the role of subtransactions from that of compensating subtransactions, we use the term *subtransaction* generically and denote an instance of either a subtransaction or a compensating subtransaction of transaction T_i as T_{ij} . Where the roles differ, we use S_{ij} to denote an instance of a subtransaction and C_{ij} to denote an instance of a compensating subtransaction. A local transaction can be thought to be composed of a single subtransaction; S_{kl} represents the subtransaction corresponding to the local transaction T_k executing at site l .

Before proceeding further, we make a distinction between a *type* of a subtransaction and an *instance* of a subtransaction. *Gtransfer1*, *Gtransfer2*, *Gaudit1*, *Gaudit2*, *Gtrans1comp* represent the different types of subtransactions in our example. Histories, on the other hand, are associated

with instances of subtransactions and compensating subtransactions. The notation T_{ij} is used to denote an instance of a subtransaction and $ty(T_{ij})$ denotes the type of a subtransaction.

To ensure that the application performs as desired when global transactions are decomposed, we develop four properties that must be satisfied by the application. But before that we develop the notion of semantic history.

Definition 1 (*Subtransactionwise serial history*). A *subtransactionwise serial history* H over a set of transactions $\mathbf{T} = \{T_1, \dots, T_m\}$ is a sequence of subtransactions and compensating subtransactions such that

1. a subtransaction T_{ij} either appears exactly once in H or does not appear at all,
2. for any two subtransactions S_{ij} and T_{ik} , S_{ij} precedes T_{ik} in H if S_{ij} precedes T_{ik} in T_i ,
3. if a compensating subtransaction $C_{ij} \in H$, then $S_{ij} \in H$ and S_{ij} precedes C_{ij} in H .
4. if S_{ij} precedes S_{im} in H and $C_{ij} \in H$, then $C_{im} \in H$ and C_{im} precedes C_{ij} in H .

Condition (1) ensures that every subtransaction or compensating subtransaction of a transaction occurs at most once. Condition (2) ensures that the order of the subtransactions in a transaction is preserved. Condition (3) ensures that a compensating subtransaction appears after the corresponding subtransaction. Condition (4) ensures that if the subtransactions of a transaction are executed in a particular order, the corresponding compensating subtransactions must be executed in the reverse order.

Definition 2 (*Complete execution*). Consider a transaction T_i decomposed into subtransactions S_{i1}, \dots, S_{in} with compensating subtransactions C_{i1}, \dots, C_{in} . The execution of T_i in a history H is a *complete* execution if either (i) all n subtransactions of T_i appear in H or (ii) some subtransactions of T_i , namely, S_{i1}, \dots, S_{ij} appear in H followed by the corresponding compensating subtransactions C_{ij}, \dots, C_{i1} .

A *semantic history* introduces state information. 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 (*Semantic history*). A *semantic history* H is a subtransactionwise serial history bound to

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

Definition 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*.

3.3. Semantic atomicity

When global transactions have been broken up into subtransactions, it may not be possible to complete a partially executed transaction. This may happen if the precondition of some later subtransaction is not satisfied and the committed subtransactions cannot be compensated for. The semantic atomicity property formalizes this idea. If an application has the semantic atomicity property, then all partially executed transactions will eventually complete. Note that if an application does not have the semantic atomicity or any of the other necessary properties described in the next few sections, then the application must be revised.⁴

Semantic atomicity property Every semantic history H_p defined over a set of transactions \mathbf{T} is a prefix of some complete semantic history H over \mathbf{T} .

Example 4. For the banking example, we must show that a partial semantic history containing one or more incomplete *Gtransfer* or *Gaudit* transaction can be completed. Consider an incomplete *Gtransfer* transaction that has completed *Gtransfer1* subtransaction. If *Gtransfer1* has successfully executed, then the precondition of *Gtrans1comp* will always be satisfied no matter what other subtransactions execute in between. *Gtrans1comp* has a precondition which checks that the account into which money is deposited is a valid account. Since this account is the same as the one involved in *Gtransfer1* and there is no transaction in the application which deletes this account, the precondition of *Gtrans1comp* is always satisfied. Next consider an incomplete *Gaudit* subtransaction that has completed *Gaudit1* but not *Gaudit2*. *Gaudit2* does not have any precondition and so it can always be executed, thereby completing *Gaudit*.

3.4. Semantic atomicity and the GTM commit protocol

The GTM commit protocol [20] classifies subtransactions as either retrievable, compensatable or pivot, and requires that subtransactions execute in the order compensatable first, pivot next, and retrievable last (see also Section 2). The following theorem relates the GTM commit protocol to our notion of semantic atomicity.

⁴ In some cases revising the application without unacceptably changing the semantics may be impossible, in which case our approach does not apply, and a more traditional, and restrictive, approach is required.

Theorem 1. *The GTM protocol [20] is more restrictive than the semantic atomicity property.*

Proof. First we show that if subtransactions are committed in the order compensatable, pivot and retrievable then we are guaranteed semantic atomicity.

Consider an incomplete transaction in a history. There can be two cases:

Case 1. Some compensatable subtransactions have committed and the other subtransactions have not yet committed. Since the compensating code does not have any preconditions, it will always be possible to complete the transaction by executing the appropriate compensating subtransactions, no matter what other subtransactions execute in between.

Case 2. The compensatable and pivot subtransactions have committed and the retrievable transactions may or may not have committed. Since retrievable subtransactions have no preconditions we are guaranteed that they can be successfully executed no matter what other subtransactions execute in between. In this way all incomplete transactions can be completed and semantic atomicity guaranteed.

Next we show by example that ordering of the GTM commit protocol is not necessary for semantic atomicity.

Consider a multidatabase composed of a hotel database and a travel agent database. Hotel rooms can be reserved directly or through the travel agent. The travel agent is consigned a number of rooms. Denote the global reserve transaction at the travel agent *ReserveT*. In subtransaction *ReserveT1*, which executes at the travel agent database, the customer is given immediate confirmation of room availability if, indeed, the number of rooms sold by the travel agent is less than the total assigned to the travel agent. In subtransaction *ReserveT2*, which executes at the hotel database, a specific room is chosen for the customer. *ReserveT2* has a precondition that the room being chosen not be assigned to another guest. *ReserveT1* is neither retrievable since it has a precondition nor compensatable since it generates an output. Therefore, *ReserveT1* is a pivot subtransaction. *ReserveT2* is compensatable but not retrievable since it has a precondition. The GTM protocol does not allow *ReserveT1* to commit before *ReserveT2*. The application, however, requires *ReserveT1* to commit before *ReserveT2* because if the hotel database is down, the customer at the travel agent's site need not wait for confirmation of the reservation.

There is also a reserve transaction at the hotel database and cancel transactions at the travel agent and at the hotel. Although details of these transactions are necessary to formally evaluate semantic atomicity, we omit them from the present discussion. Instead we state without proof that *ReserveT1* may commit before *ReserveT2* since it is the case that the precondition of *ReserveT2* will eventually be satisfied, no matter what other transactions interleave. Thus the semantic atomicity property holds. \square

4. Applications with global integrity constraints

In applications without global integrity constraints, the semantic atomicity property is enough to ensure semantic correctness. However, some multidatabase applications have global integrity constraints, in which case the execution of a subtransaction may no longer preserve the global integrity constraints. In such cases we need assurance that users are not displayed inconsistent data and eventually the database returns to a consistent state. This motivates us to propose the sensitive transaction isolation property and the consistent execution property. Before describing these properties, we extend the banking example to illustrate the complexities introduced in the decomposition process because of global integrity constraints.

Example 5 (*Extended banking enterprise example*). The multidatabase banking application presented in Example 1 is upgraded to include a new object in the head office database which stores the sum of account balances in all the branches and is denoted by *banktotal*.

The deposit and withdraw transactions now must update the object *banktotal*. This means a branch, except for the head office, must process deposit and withdraw transactions as global transactions. Let *GDeposit*, *GWithdraw* be the global transactions executed by a branch to deposit, withdraw money, respectively. *GDeposit* is decomposed into two subtransactions of type *GDeposit1* and *GDeposit2*, where subtransactions of type *GDeposit1* execute at the head office and those of type *GDeposit2* execute at the branch. *GDeposit1* has no preconditions; the postcondition of *GDeposit1* increments *banktotal* by the amount deposited. *GDeposit2*'s precondition is the account into which money is deposited must be a valid account; *GDeposit2*'s postcondition increments the balance of the account by the amount of money deposited. *GWithdraw* is also decomposed into two subtransactions of type *GWithdraw1* and *GWithdraw2*, where subtransactions of type *GWithdraw1* execute at the head office and those of type *GWithdraw2* execute at the branch. *GWithdraw1* has a precondition that *banktotal* must exceed the amount of money to be withdrawn. The postcondition of *GWithdraw1* decrements *banktotal* by the amount of money to be withdrawn. *GWithdraw2* has two preconditions: the account must be a valid account and the account must have sufficient funds. The postcondition of *GWithdraw2* decrements the balance of the account by the amount withdrawn. Since the head office stores the object *banktotal*, it processes the withdraw and deposit transactions as local transactions, denoted by *LWithdraw* and *LDeposit*. The preconditions of *LWithdraw* ensure that the account from which money is to be withdrawn is a valid account having sufficient funds. The postconditions of *LWithdraw* decrement the balance of the account and the object *banktotal*, each by the amount of money withdrawn. *LDeposit* has a precondition, namely, the account into which money is deposited must be a

valid account. The postconditions increment the balance of the account as well as the object *banktotal*, each by the amount of money deposited. The transfer transactions do not affect the object *banktotal* and are specified as in Example 1. *LTransfer*, *GTransfer* denote the local, global transfer transaction, respectively. *GTransfer* is decomposed into *GTransfer1* and *GTransfer2* where *GTransfer1* withdraws money from an account and *GTransfer2* deposits money into an account. The *GAudit* transaction is composed of two subtransactions of type *GAudit1* and *GAudit2* where subtransactions of type *GAudit1* execute at the head office and those of type *GAudit2* execute at the branch. *GAudit1* has no preconditions. The postcondition of *GAudit1* prints the value of the objects *banktotal* and *accbal*. *GAudit2* has no preconditions and produces as output the value of the object *accbal*.

Corresponding to each type of global subtransaction (except the types of subtransactions which occur last in a transaction) we specify a compensating subtransaction. The compensating subtransactions corresponding to *GDeposit1*, *GWithdraw1*, *GTransfer1* are denoted by *GDep1Comp*, *GWith1Comp*, *GTrans1Comp*, respectively. The specifications of *GDep1Comp*, *GWith1Comp*, *GTrans1Comp* are similar to that of *GWithdraw1*, *GDeposit1*, *GTransfer2*, respectively. *GAudit1* generates an output, and thus has no compensating subtransactions.

For this extended example also we take the instance of the banking enterprise consisting of a branch (Branch *B*) and a head office (Head Office *H*). The banking enterprise has the following global integrity constraint:

$$H.banktotal = \sum B.accbal + \sum H.accbal,$$

where $\sum B.accbal$, $\sum H.accbal$ represent the sum of account balances in Branch *B* and *H*, respectively.

A formal specification of this example is given in Appendix A.

4.1. Generalizing the global integrity constraints

When global transactions have been decomposed into atomic subtransactions, the global integrity constraints may not hold after the execution of a subtransaction.

Example 6. Consider the *GTransfer* transaction that has been decomposed into *GTransfer1* and *GTransfer2*, where *GTransfer1* decrements the balance of the account from which money is withdrawn and *GTransfer2* increments the balance of the account into which money is deposited. After the execution of *GTransfer1*, the integrity constraint – *banktotal* equals the sum of account balances in all the branches – does not hold.

Execution of global subtransactions may lead to an inconsistent database state. Other subtransactions might be exposed to this inconsistency resulting in undesirable consequences. Once the global integrity constraints have been violated we have no formal basis for assessing the correctness of executions.

To solve this problem, we relax the global integrity constraints to get a new set of generalized integrity constraints such that this new set of constraints is satisfied before and after the execution of each subtransaction [2]. The number of integrity constraints that must be generalized depends on the number of integrity constraints violated by the decomposition of global transactions. Note that global integrity constraints can always be generalized irrespective of the domains of the variables in the constraint. We use auxiliary variables to generalize the integrity constraints. Auxiliary variables are a standard method of reasoning about concurrent executions [22] and have been applied to the problem of semantic-based concurrency control [2,12]. Auxiliary variables are introduced for the purpose of analysis only and our goal is to exclude such variables in the implementation.

Example 7. We associate an auxiliary variable with each type of subtransaction of a global transaction. These auxiliary variables are natural numbers which are incremented each time the corresponding subtransaction executes. For example, *tempDep1*, *tempDep2*, *tempDep1Comp* are the auxiliary variables associated with *GDeposit1*, *GDeposit2* and *GDep1Comp* type of subtransaction, respectively. Each time *GDeposit1* executes, it increments the variable *tempDep1* by the amount of money to be deposited. Similarly, the variables *tempDep2*, *tempDep1Comp* are incremented by *GDeposit2*, *GDep1Comp*, respectively.

Note that it is possible to reduce the number of auxiliary variables by allowing two or more subtransactions to update the same auxiliary variable. Our approach in which an auxiliary variable is updated by one type of subtransaction has the following advantages. Firstly, it helps to capture whether there are any incomplete transactions at any given state. Secondly, the behavior of auxiliary variable is easy to understand which helps in building the model that is to verified by the model checker.

The generalized integrity constraint for the banking enterprise consisting of the Branch *B* and Head Office *H* is as follows:

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

The above expression is written in `Object Z`. It indicates that $H.banktotal$ equals sum of balances in Branch B and Head Office H plus an expression involving the auxiliary variables.

4.2. Consistent execution property

Our model presented so far allows subtransactions to be executed when the generalized integrity constraints are satisfied. However, after all the subtransactions complete, the database must be in a consistent state. To ensure this we place the following restriction on semantic histories.

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

Example 8. The banking enterprise will be in a consistent state when the original integrity constraint $H.banktotal = \sum B.accbal + \sum H.accbal$ is satisfied.

In the initial state before any transaction has executed, all the auxiliary variables are initialized to zero and the database is in a consistent state.

Consider a $GDeposit$ transaction. The subtransaction $GDeposit1$ executing at the head office increments $H.tempDep1$ by the amount to be deposited. If the corresponding $GDeposit2$ executes at the branch B , the auxiliary variable $B.tempDep2$ is incremented by the same amount; otherwise if $GDep1Comp$ executes at the head office the auxiliary variable $H.tempDep1Comp$ is incremented. Thus when all $GDeposit$ transactions complete, the following predicate holds:

$$H.tempDep1 = B.tempDep2 + H.tempDep1Comp.$$

Similarly, when all $GTransfer$ transaction initiated at the head office completes, we have

$$H.tempXfer1 = B.tempXfer2 + H.tempXer1Comp.$$

We can write similar predicates for the other types of transactions.

To ensure that the database eventually returns to a consistent state we place the following temporal requirement as a history invariant in the specification.

$$\begin{aligned} & \square(((H.tempDep1 = B.tempDep2 + H.tempDep1Comp) \\ & \quad \wedge (H.tempWith1 = B.tempWith2 + H.tempWith1Comp) \\ & \quad \wedge (H.tempXfer1 = B.tempXfer2 + H.tempXer1Comp) \\ & \quad \wedge (B.tempXfer1 = H.tempXfer2 + B.tempXer1Comp)) \\ & \Rightarrow (H.banktotal = \sum B.accbal + \sum H.accbal)) \end{aligned}$$

The notations \square stand for ‘always’. The property states that it is always true that when all transactions complete, the database satisfies the original integrity constraints.

4.3. Sensitive transaction isolation property

In our model, we allow subtransactions or transactions to see database states that do not satisfy the original integrity constraints (i.e., states satisfying the generalized integrity constraints but not the original integrity constraints). 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 be viewed by users or to be used by some processes; these transactions are referred to as *sensitive* transactions by Garcia–Molina [12]. 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 (a state satisfying the original integrity constraints), even though T_i may be running on an inconsistent database state (that is, a state satisfying the generalized integrity constraints and not satisfying the original integrity constraints).

In our model, the goal is to 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 true before the subtransactions of the sensitive transaction are executed. One approach [2] for enforcing this is to include additional preconditions in the subtransactions of sensitive transactions; these preconditions must imply the subset of the original integrity constraints that must hold before the execution of the subtransactions. Note that these preconditions in a subtransaction may involve non-local variables, which means that in order to verify these preconditions a subtransaction may have to read from a non-local database. Since a subtransaction executes atomically, having non-local reads will require the support for a global commit protocol which in turn will violate local autonomy. Hence this approach is not feasible in a multidatabase environment. The solution is to enforce the sensitive transaction isolation property using history invariants. Second, as pointed out by Rastogi et al. [24], if outputs are generated by multiple subtransactions, interleavings must be controlled to ensure that outputs from later subtransactions are consistent with outputs from earlier subtransactions.

Example 9. The banking example has one sensitive transaction, namely, *GAudit*. *GAudit* consists of two subtransactions of type *GAudit1* and *GAudit2*.

In the first part, we compute the global integrity constraints that must be satisfied before any subtransaction of type *GAudit1* or *GAudit2* can be executed. As it turns out, the global integrity constraint $H.banktotal = \sum H.accbal + \sum B.accbal$ must be satisfied before subtransactions of types *GAudit1* or *GAudit2* can be executed. The first aspect of the sensitive transaction isolation property, which ensures that subtransactions of types *GAudit1* and *GAudit2* are always executed in a consistent state, is captured by the following history invariant.

$$\square((H.banktotal \neq \sum B.accbal + \sum H.accbal) \Rightarrow ((\bigcirc op \neq GAudit1) \wedge (\bigcirc op \neq GAudit2))).$$

The notation ' $\bigcirc op$ ' stands for the next operation. The property states that it is always true that when the global integrity constraint is not satisfied, the next operation cannot be a subtransaction of type *GAudit1* or *GAudit2*.

In the second part, we control the interleaving of subtransactions of global transactions that is involved in changing the objects involved in the computation of outputs produced by *GAudit*. The goal is to control the interleaving such that the two outputs produced by *GAudit1* and the corresponding *GAudit2* appear to come from the same global state. *GAudit1* outputs the local state of the head office and *GAudit2* outputs the local state of the branch. We compute the global transactions that changes the objects involved in the computation of the outputs produced by *GAudit1* as well as the objects involved in the computation of outputs produced by *GAudit2* and disallow such transactions from interleaving between *GAudit1* and *GAudit2*. For example, a *GTransfer* transaction consisting of subtransactions of types *GTransfer1* and *GTransfer2* change the objects involved in producing the outputs of *GAudit1* and *GAudit2*. Thus *GTransfer1* and *GTransfer2* should not be allowed to interleave between *GAudit1* and *GAudit2*. Note however that the unsuccessful execution of *GTransfer* (that is, *GTransfer1* followed by *GTrans1Comp*) changes only one local state, and is not a problem. Thus we disallow only *GTransfer2* from executing between the subtransactions of *GAudit*. For the same reason, we disallow *GDeposit2*, *GWithdraw2* from executing between the subtransactions of *GAudit*.

We use the auxiliary variables *tempAudit1* and *tempAudit2* to ensure this aspect of the sensitive transaction isolation property. When a subtransaction of type *GAudit1* executes, it increments *tempAudit1*. Similarly, *GAudit2* increments the variable *tempAudit2* each time it executes. The second aspect of the sensitive transaction isolation property is therefore captured by the following history invariant.

$$\square((H.tempAudit1 \neq B.tempAudit2) \Rightarrow ((\bigcirc op \neq GDeposit2) \wedge (\bigcirc op \neq GWithdraw2) \wedge (\bigcirc op \neq GTransfer2))).$$

The above formalism indicates that if $H.tempAudit1$ is not equal to $B.tempAudit2$, then the next operation must not be a $GDeposit2$ or a $GWithdraw2$ or a $GTransfer2$ operation.

After giving the necessary properties required for correct execution, we are now ready to give a definition of a correct semantic history.

Definition 5 (*Correct semantic history*). A semantic history generated from the specification satisfying all the three properties, that is, the semantic atomicity property, the consistent execution property and the sensitive transaction property, is defined to be a *correct* semantic history.

5. Developing an efficient mechanism

The previous section outlines how global transactions can be decomposed, and how the interleaving of the decomposed transactions must be controlled to avoid inconsistencies. Although the decomposed specifications are helpful for analysis, it is undesirable to directly implement the decomposed specifications. First, it is expensive to implement the auxiliary variables. Second, the history invariants must be checked before an operation can be dispatched. The checking of history invariants is an expensive operation as it involves reading several databases before submitting an operation. Also the checking and the actual operation must be done as an atomic action which in turn requires support for an atomic commitment protocol, which contradicts our primary goal. Thus our objective is to control the interleaving without implementing the auxiliary variables or checking the history invariants. Successor set is the mechanism we use to achieve this objective.

5.1. Successor set mechanism

The successor set mechanism [2] allows us to control the interleaving of subtransactions of different global transactions. However, unlike the successor set mechanism proposed in [2], the successor set mechanism in a multidatabase application is distributed in nature. Each site stores a successor set corresponding to each of the different types of subtransactions processed by that site.

Definition 6 (*Successor Set*). The successor set of a type of subtransaction is a set of types of subtransactions. The successor set of type of T_{ij} is denoted by $SS(ty(T_{ij}))$.

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.

To formalize the set of allowable interleavings, we define a correct successor set history.

Definition 7 (*Correct successor set history*). Let H be a correct semantic history. If T_i is incomplete in the prefix of H that ends at T_{pq} , and T_{ij} is the last subtransaction in T_i such that

1. T_{ij} conflicts with T_{pq} and
2. T_{ij} precedes T_{pq} in H

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

Note that we define successor set for each type of step (not for each type of transaction). This definition is intended to allow for maximum concurrency. For example, suppose a global transaction T_i is decomposed into three subtransactions T_{ij} , T_{ik} , and T_{il} . Say subtransaction T_{pq} is not allowed to execute between T_{ij} and T_{ik} but it is allowed to execute between T_{ik} and T_{il} . Excluding $ty(T_{pq})$ from $SS(ty(T_{ij}))$ and including it in $SS(ty(T_{ik}))$ will help to achieve the above interleaving. However, if successor sets are associated with each type of transaction, then T_{pq} will not be in the successor set of T_i and it cannot execute until T_i completes.

With respect to the specifications given with history invariants not all successor set descriptions are valid. Informally, a successor set is valid with respect to the specification if any correct successor set history is equivalent to some correct history generated from the specification. 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 history invariants if

1. any correct successor set history generated by S_2 is equivalent to a correct semantic history generated by S_1 , and
2. S_2 satisfies the semantic atomicity property.

Example 10. A possible successor set description for the banking example is given below. The Head Office H stores the following successor set description.

$$\begin{aligned}
SS(GWithdraw1) &= SS(GDeposit1) = SS(GTransfer1) \\
&= \{GWithdraw1, GDeposit1, GTransfer1, GTransfer2, \\
&\quad GTrans1Comp, GWith1Comp, GDep1Comp\}
\end{aligned}$$

$$SS(GAudit1) = \{GAudit1\}.$$

The Branch *B* stores the following successor set description.

$$\begin{aligned}
SS(GTransfer1) &= \{GWithdraw2, GDeposit2, GTransfer1, \\
&\quad GTransfer2, GTrans1Comp\}.
\end{aligned}$$

The successor set description at the head office imposes the following restriction. When a subtransaction of type *GAudit1* has executed, no subtransactions of type *GWithdraw1*, *GDeposit1*, *GTransfer1*, *GTransfer2*, *GDep1Comp*, *GWith1Comp* or *GTrans1Comp* can execute at the head office until the *GAudit* transaction completes. On the other hand, if a *GWithdraw1*, *GDeposit1* or a *GTransfer1* type of subtransaction has been executed, no subtransaction of types *GAudit1* or *GAudit2* can execute until the *GWithdraw*, *GDeposit* or *GTransfer* transaction completes.

The successor set description at the branch imposes the restriction that if a subtransaction of type *GTransfer1* has been executed, it will not be possible to execute a subtransaction of type *GAudit2* until the transfer transaction completes.

We do not specify successor sets for the types of subtransactions that occur last in the transaction; when the last subtransaction completes, the transaction terminates, and any type of subtransaction can execute. Appendix B contains a proof of the valid successor set property for the banking enterprise example.

6. Support for global transactions

A correct successor set history is a semantic history which in turn is a subtransactionwise serial history. For every pair of operations in a subtransactionwise serial history, all operations of one subtransaction must appear before any operation of the other subtransaction. However, if the subtransactions of a transaction execute atomically and without any interleaving, the database makes poor use of system resources. The standard solution, which we adopt, is to increase the class of allowable histories to include the histories in which subtransactions need not be executed serially, but nevertheless whose effect is the same as that of a correct successor set history.

Before giving the definition of subtransactionwise serializable histories we define history in a manner similar to that defined by Bernstein [5].

Definition 8 (History). A history H defined over a set of transactions $\mathbf{T} = \{T_1, \dots, T_m\}$, where each transaction T_i executes in sites denoted by the set $Site_i$, is a partial order of operations with ordering relation \prec_H where:

1. $H = \bigcup_{i=1}^m \bigcup_{j \in Site_i} T_{ij}$
2. $\prec_H \supseteq \bigcup_{i=1}^m \bigcup_{j \in Site_i} \prec_{ij}$; 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 subtransactions of \mathbf{T} . Condition (2) says that H preserves the order of operations in each subtransaction. Condition (3) says that every pair of conflicting operations are ordered in H .

Definition 9 (Subtransactionwise serializable history). A subtransactionwise serializable history is one that is conflict equivalent to a correct successor set history.

After describing our notion of correct concurrent execution, we now describe a mechanism for supporting global transactions. For the purpose of coordinating the actions of the global transactions we need global transaction managers. However, unlike other approaches [20,24], we do not require a centralized global transaction manager. Each site supports a global transaction manager for coordinating the global transactions and subtransactions submitted at that site. A global transaction manager in our model performs two functions. First, it produces subtransactionwise serializable histories. Second, it coordinates the global transactions submitted at its site and achieves semantic atomicity. In the following sections we discuss how the global transaction managers carry out these functions.

6.1. Producing subtransactionwise serializable executions

We assume that each local DBMS supports some concurrency control protocol which produces serializable executions of the local transactions and global subtransactions submitted at that site. The global transaction manager at each site stores the successor set description of all types of subtransactions that can execute at that site. When a global transaction manager receives a subtransaction which is to be executed at that site, it checks whether the subtransaction's type is in the successor set of the types of the committed subtransactions of active transactions. If so, it forwards the subtransaction for execution to the local DBMS. Otherwise, the global transaction manager has to wait for the completion of some other transaction before issuing the subtransaction to the local DBMS. The local DBMS executes the subtransaction; if the execution is successful it commits the subtransaction, otherwise it aborts the subtransaction.

6.2. Achieving semantic atomicity

The global transaction manager becomes the coordinator of the transactions submitted at that site. The global transaction manager after receiving the transaction distributes the subtransactions to the other sites in an order consistent with the ordering of the subtransactions of the global transaction. After a subtransaction at a site completes, the global transaction manager at that site informs the coordinator of the outcome which then takes appropriate action. For example, if a subtransaction aborts, the coordinator might resubmit the subtransaction or issue compensating subtransactions to the other sites. Note that the semantic atomicity property ensures that at least one of the above approaches will be feasible. When the global transaction terminates, the coordinator sends a termination message to all the other sites executing this transaction. Note that we do not address multidatabase recovery issues in this paper and refer the interested reader to the work by Georgakopoulos [11].

6.3. Proof of correctness

We assume that each local DBMS supports some concurrency control protocol which generates serializable schedules. From the previous two sections we know that the global transaction managers control the scheduling in the following way.

1. If T_{ij} must precede T_{ik} of some transaction T_i , then T_{ik} is dispatched to the local scheduler only after T_{ij} completes.
2. If $ty(T_{pq}) \notin SS(ty(T_{iq}))$, then T_{pq} is not dispatched to the local scheduler until all subtransactions of T_i complete.

Theorem 2. *All complete histories generated by the above mechanism are subtransactionwise 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 subtransactions 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 subtransactions in each H_i . (b) For each pair of global subtransactions T_{ij} and T_{pq} such that the global subtransaction T_{ij} is submitted for execution after the execution of the global subtransaction 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 subtransactionwise serial, and (ii) if T_i is

incomplete in the prefix of H_s that ends in T_{pj} such that T_{ij} 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 subtransactions of each transaction T_i in H_s . Since the global transaction manager submits the subtransactions for execution in an order consistent with the ordering of subtransactions in a transaction and this ordering is preserved in H_s , H_s is subtransactionwise serial.

For (ii), assume the contrary. Suppose T_i is incomplete in the prefix of history H_s ending in T_{pj} , T_{ij} 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 subtransactions 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.

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 subtransactions 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 subtransactionwise serializable. \square

7. Automated verification of the decomposed specifications

To use our approach on real world applications, automated verification of the properties given in Sections 3 and 4 is desirable. Object Z does not have the tool support necessary to discharge the required proof obligations automatically. Even if Object Z did have state-of-the-art tools, theorem proving is quite difficult and far from ‘automatic’. The limited success of theorem proving, coupled with dramatic increase in the capabilities of model-checking systems, has prompted researchers to advocate model checking as an alternative to theorem proving [26]. Model checking has long been used successfully in hardware verification, and there are now enough examples of its success for software to suggest applying model checking here (for example, [1,8]).

A model checker requires that the system being verified be represented as a finite state machine. The model checker then performs an exhaustive search of the state space to see whether properties specified as temporal formulae hold. Most software systems are infinite state machines, and consequently a model checker cannot directly verify such systems. To solve this problem, researchers [26,15] propose developing finite state abstractions of the software system which can be verified by model checkers. In this section we describe how we develop one finite model of the banking application (many others are, of

course, possible). We verify the abstraction using the SMV model checker, which may be downloaded without charge from a website at Carnegie Mellon University.⁵

7.1. 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 change 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.

7.2. Finite state abstraction of the banking example

Our basic goal is to show that encoding an abstraction of a multidatabase specification for a model checker requires roughly the same level of effort as encoding such a specification in a programming language. Special purpose tools could undoubtedly speed the encoding process. In other words, the verification of the relevant properties does not appear infeasible. In the following paragraphs we describe simple finite models of the banking enterprise presented in Example 5 with a branch *B* and a head office *H*.

State variables. The state variables representing database objects or auxiliary variables are either associated with the head office *H* or the branch *B*. The variables associated with the head office are prefixed with *H* and those associated with the branch are prefixed with *B*. The branch *B* and head office *H* each have two accounts; the account id's are integers which can take the value 1 or 2. Thus, the account id's which are input to the transactions, denoted by *acc*, *acc1*, *acc2*, are specified as integers in the range 1–2. The state variable

⁵ The website is <http://www.cs.cmu.edu/afs/cs/project/modck/pub/www/modck.html>.

Baccbal stores the account balance information in branch *B*. *Baccbal* is specified as a fixed array; *Baccbal*[*x*] gives the balance of account *x* in branch *B*. Each transaction can modify an account balance by an amount equal to 1 only. Each account balance can take any value from 0 to 100. We specify *Haccbal*, which stores account balances of accounts in head office *H*, in a similar manner. The state variable *Hbanktotal* stores the total money stored in all the accounts in branch *B* and head office *H*. Since each account can take values 0 or 100 and there are four accounts in the banking enterprise, *Hbanktotal* is specified as an integer whose range is from 0 to 400. Note that our effort in minimizing the execution time required by the model checker has prompted our choice of values for the variables. We believe that scaling the values does not affect significantly the validity of our results, since many faults can be found with ‘simple’ test cases.⁶ However, it is clearly possible that there might be faults that can only be discovered with larger values for the variables.

Besides the database variables, we also have auxiliary variables in the Object Z specification. These auxiliary variables are associated with subtransactions of global transactions. In the finite models we have a state variable corresponding to each auxiliary variable. Examples include the state variables *HtempAudit1*, *BtempAudit2*, *HtempWith1*, and *BtempWith2*. Note that these variables also keep track of the number of subtransactions of the corresponding type executed so far. For example, the value of *HtempAudit1* indicates the number of *GAudit1* transactions executed. Each of these state variables must be specified with a range. The upper limit, specified using another state variable, gives the maximum number of allowable subtransactions of a given type. For example, *maxAudit* gives the maximum number of *GAudit1* or *GAudit2* subtransactions. Thus, *HtempAudit* is specified with a range of 0 to *maxAudit*. By controlling the upper limit of this range (that is, the value of *maxAudit*) we are able to vary the number of subtransactions and get different models. For example, one model may set *maxAudit* to 0 and the other may set it to 10. Note that each of these models can handle only a finite number of transactions.

To verify the properties we need some state variables which act as flags to indicate when a subtransaction has been executed. For example, the state variable *HGAudit1JustExecuted* is a variable which can take any of the two values *Y* and *N*. When a *HGAudit1* subtransaction executes, the value of *HGAudit1JustExecuted* = *Y*, otherwise *HGAudit1JustExecuted* = *N*.

We have one variable *input* which enumerates all possible types of subtransactions and compensating subtransactions. The local transactions of branch *B* are prefixed with *BL*, and those of head office *H* are prefixed with *HL*. Similarly, the global subtransactions of branch *B* are prefixed with *BG* and

⁶ Nitpick is a counterexample generator build on this philosophy [15].

those of head office are prefixed with *HG*. Since the *LAudit* transaction does not change state and it is not interesting from the analysis perspective, we omit the *LAudit* transaction from the finite model.

7.2.1. State initialization

The initial state of each variable is specified by the *init* function. The states of all auxiliary variables are initialized to 0. The *Hbanktotal*, *Haccbal*[1], *Haccbal*[2], *Baccbal*[1] and *Baccbal*[2] are all initialized to 0. The variables representing the flags whether a transaction has executed or not are initialized to *N*.

7.2.2. State transformation

The value of the variable is changed according to the *next* function. The next function is specified by a case expression. Each subtransaction which updates the variable usually contribute one case statement in the case expression. The left-hand side of the colon (:) is an expression which is conjunction of the following: (i) the name of the subtransaction activating the state change, (ii) the preconditions of the subtransactions, (iii) if no direct assignment is made to the variable, extra preconditions to check that the variable lies within the specified range, (iv) preconditions involving auxiliary variables to ensure the correct sequence in which the subtransaction must be executed, (v) preconditions involving auxiliary variables to ensure that the number of subtransactions of that type has not exceeded the upper limit. The right-hand side of the colon (:) specifies how the variable must be updated if the left-hand side expression is satisfied. The last statement in the case expression specifies the default value of the variable in case none of the previous cases has been satisfied. In our model, the left hand side of the last case statement is 1, and the right-hand side specifies the current value of the variable. The initialization and state transformation of *Haccbal*[1] are given in Fig. 1. The first line states that the initial state of *Haccbal*[1] is equal to 0. The subsequent lines describe under what conditions the variable *Haccbal*[1] may be changed. For example, the two lines following the *case* word describes that the variable *Haccbal*[1] is changed when the input is *HLTransfer*, the money is withdrawn from *acc1* and deposited into *acc2*, *amt* to be withdrawn is less than or equal to *Haccbal*[1], the *amt* when deposited in *acc2* should not exceed *maxacctbal*, number of local transactions in the head office is less than the maximum allowed.

7.3. Specifying the invariants and properties using CTL

7.3.1. Generalized invariants

The mapping of generalized invariant in Object Z to the corresponding CTL formula is straightforward as shown below.

```

init(Haccbal[1]) := 0;
next(Haccbal[1]) := case
  (input = HLTransfer) & (acc1 = 1) & (acc2 = 2) & (amt <= Haccbal[1])
    & (Haccbal[2] + amt <= maxacctbal) & (HLXferCount < maxHLXfer) : Haccbal[1] - amt;
  (input = HLTransfer) & (acc1 = 2) & (acc2 = 1) & (amt <= Haccbal[2])
    & (Haccbal[1] + amt <= maxacctbal) & (HLXferCount < maxHLXfer) : Haccbal[1] + amt;
  (input = HLDeposit) & (acc = 1) & (Haccbal[1] + amt <= maxacctbal)
    & (amt + Hbanktotal <= maxbanktotal) & (LDepCount < maxLDep) : Haccbal[1] + amt;
  (input = HLWithdraw) & (acc = 1) & (amt <= Haccbal[1]) & (amt <= Hbanktotal)
    & (LWithCount < maxLWith) : Haccbal[1] - amt;
  (input = HGTransfer1) & (acc1 = 1) & (amt <= Haccbal[1]) &
    (HtempXfer1 + amt <= maxHXfer) : Haccbal[1] - amt;
  (input = HGTransfer2) & (acc2 = 1) &
    (BtempXfer1 > (HtempXfer2 + BtempXfer1Comp)) &
    (HtempXfer2 + amt <= maxBXfer) &
    ((HtempAudit1 = BtempAudit2)) &
    (Haccbal[1] + amt <= maxacctbal) : Haccbal[1] + amt;
  (input = HGTrans1Comp) & (acc1 = 1) &
    (HtempXfer1 > (HtempXfer1Comp + BtempXfer2)) &
    (HtempXfer1Comp + amt <= maxHXfer) &
    (amt + Haccbal[1] <= maxacctbal) : Haccbal[1] + amt;
  1 : Haccbal[1];
esac;

```

Fig. 1. Initialization and state transitions of $Haccbal[1]$.

$$\begin{aligned}
AG(&Hbanktotal = Baccbal[1] + Baccbal[2] + Haccbal[1] + Haccbal[2] \\
&- HtempWith1 + BtempWith2 + HtempWith1Comp \\
&+ HtempDep1 - BtempDep2 - HtempDep1Comp \\
&+ BtempXfer1 - HtempXfer2 - BtempXfer1Comp \\
&+ HtempXfer1 - BtempXfer2 - HtempXfer1Comp).
\end{aligned}$$

Recall that a CTL formula with the prefix AG means ‘for all paths, for all states’. The model checker was able to verify this property indicating that all states in the finite model satisfied the generalized invariant.

7.3.2. Consistent execution property

The consistent execution property is specified in CTL as follows:

$$\begin{aligned}
AG(&(HtempXfer1 = BtempXfer2 + HtempXfer1Comp) \& (BtempXfer1 \\
&= HtempXfer2 + BtempXfer1Comp) \& (HtempDep1 \\
&= BtempDep2 + HtempDep1Comp) \& (HtempWith1 \\
&= BtempWith2 + HtempWith1Comp) - > (Hbanktotal \\
&= Baccbal[1] + Baccbal[2] + Haccbal[1] + Haccbal[2])).
\end{aligned}$$

The model checker verified this property ensuring that when all transactions complete the resulting database state is consistent.

7.3.3. Sensitive transaction isolation property

There are two aspects to the sensitive transaction isolation property. The first aspect requires that no subtransaction of *GAudit* must execute when the original integrity constraints do not hold. Recall that the flags *HGAudit1JustExecuted* and *BGAudit2JustExecuted* indicate whether the subtransactions *HGAudit1* and *BGAudit2* have just been executed or not. This aspect of the sensitive transaction isolation property is specified in CTL as follows:

$$AG(\neg(Hbanktotal = Baccbal[1] + Baccbal[2] + Haccbal[1] + Haccbal[2]) \rightarrow ((HGAudit1JustExecuted = N) \& (BGAudit2JustExecuted = N))).$$

The expression above indicates that it is always the case that when the global integrity constraint is not satisfied, the operations *HGAudit1* or *BGAudit2* are not executed. The model checker verified the existence of this property.

The second aspect of the sensitive transaction isolation property prohibits the subtransactions *BGDeposit2*, *BGWithdraw2* and *BGTransfer2* from executing when the global *GAudit* transaction is executing. This property is formalized in CTL as follows:

$$AG(\neg(HtempAudit1 = BtempAudit2) \rightarrow ((BGTransfer2JustExecuted = N) \& (BGDeposit2JustExecuted = N) \& (BGWithdraw2JustExecuted = N) \& (HGTransfer2JustExecuted = N))).$$

The model checker verified this property as well ensuring the satisfaction of sensitive transaction isolation property.

7.3.4. Semantic atomicity property

The semantic atomicity property requires that it be possible to extend any partial semantic history to a complete semantic history without initiating a new transaction. For each of the models we have built we show that if the preconditions of none of the later subtransactions (subtransactions that are not the first subtransaction of global transactions) are satisfied, then all the transactions have completed execution.

Below we express the semantic atomicity property using the CTL logic. We have seven types of subtransactions that execute only after the corresponding first subtransaction has executed. The expression before the $->$ symbol describes the condition that if none of the preconditions of these seven later subtransactions are satisfied. The part after the $->$ symbol indicates the condition that all subtransactions have completed execution.

$$\begin{aligned}
& AG((!(HtempDep1 > (BtempDep2 + HtempDep1Comp)) \\
& \quad \&(banktotal \geq amt) \\
& \quad \&(HtempDep1Comp + amt \leq maxGDep))\& \\
& !((HtempWith1 > (HtempWith1Comp + BtempWith2)) \\
& \quad \&(banktotal + amt \leq maxbanktotal) \\
& \quad \&(HtempWith1Comp + amt \leq maxGWith))\& \\
& !((HtempXfer1 > (HtempXfer1Comp + BtempXfer2))\& \\
& \quad (HtempXfer1Comp + amt \leq maxHXfer\& \\
& \quad ((Haccbal[2] + amt \leq maxacctbal)|(Haccbal[1] \\
& \quad + amt \leq maxacctbal)))\& \\
& !((BtempXfer1 > (BtempXfer1Comp + HtempXfer2))\& \\
& \quad (BtempXfer1Comp + amt \leq maxAXfer)\& \\
& \quad ((Baccbal[2] + amt \leq maxacctbal)|(Baccbal[1] \\
& \quad + amt \leq maxacctbal)))\& \\
& !((HtempXfer1 > (BtempXfer2 + HtempXfer1Comp))\& \\
& \quad (HtempAudit1 = BtempAudit2)\&(BtempXfer2 + amt \leq maxHXfer)\& \\
& \quad (((acc2 = 1)\&(Baccbal[1] + amt \leq maxacctbal))| \\
& \quad ((acc2 = 2)\&(Baccbal[2] + amt \leq maxacctbal))))\& \\
& !((BtempXfer1 > (HtempXfer2 + BtempXfer1Comp))\& \\
& \quad (HtempAudit1 = BtempAudit2)\&(HtempXfer2 + amt \leq maxAXfer)\& \\
& \quad (((acc2 = 1)\&(Haccbal[1] + amt \leq maxacctbal))| \\
& \quad ((acc2 = 2)\&(Haccbal[2] + amt \leq maxacctbal))))\& \\
& !((HtempDep1 = BtempDep2 + HtempDep1Comp)\& \\
& \quad (BtempXfer1 = HtempXfer2 + BtempXfer1Comp)\& \\
& \quad (HtempWith1 = BtempWith2 + HtempWith1Comp)\&(BtempAudit2 < HtempAudit1) \\
& \quad \&(BtempAudit2 < maxAudit))) \\
& -> ((HtempDep1 = BtempDep2 + HtempDep1Comp)\& \\
& \quad (HtempWith1 = BtempWith2 + HtempWith1Comp)\& \\
& \quad (HtempXfer1 = BtempXfer2 + HtempXfer1Comp)\& \\
& \quad (BtempXfer1 = HtempXfer2 + BtempXfer1Comp)\&(BtempAudit2 = HtempAudit1))).
\end{aligned}$$

The model checker verified this property for all the models that we developed.

7.4. Discussion

The finite models introduced some constraints not present in the `Object Z` specification. One such constraint is placing an upper limit on the values of state variables. For example, the `Object Z` specification has no upper limit on the values of `accbal` or `banktotal` but the corresponding state variables in the finite model do have one. This in turn introduces additional preconditions which check that the variable does not exceed the upper limit. Adding the extra preconditions on the subtransactions in the finite state model destroys the semantic atomicity property. To overcome this problem, we limit the total number of transactions such that the upper limit of `accbal` or `banktotal` will never be reached.

The time required to verify a model did not vary much with the number of transactions allowed by the model. For example, the time required to verify a model consisting of one transaction was 5.33 seconds on an Ultra-30 running Solaris 2.6 and that required to verify a model consisting of nine transactions was 5.67 seconds. However, the total time required to generate and verify all the models of nine transactions was quite significant (6.5 days) because of the large number (24 310) of models generated.

Our model checking verification does not show that the properties hold in all reachable states in the `Object Z` specification. Instead, it shows that the properties hold for a certain finite abstraction of the `Object Z` specification. We can characterize our analysis by stating with certainty that there is no fault with respect to the properties such that the fault can be triggered in the particular model that we investigated. While less than full verification through theorem proving, such a statement is certainly much stronger than can be obtained through any conventional testing approach.

8. Conclusion

Our contribution in this paper is to define a model for multidatabases in which the four goals of local autonomy, distributed management of global transactions, maintenance of integrity constraints, and correctness of execution histories are simultaneously achieved. We argue that no other multidatabase proposal meets all of these goals simultaneously as well as our proposal. We begin with a semantic notion of correctness composed of three properties: semantic atomicity, consistent execution, and sensitive transaction isolation. These three properties yield two goals: maintaining

integrity constraints and correctness of execution histories. We manage global transactions in a distributed fashion with a successor set mechanism at each site. Successor set descriptions are shown to be valid refinements of specifications via the fourth of our properties, namely the valid successor set property. The successor set mechanism yields the remaining two goals: local autonomy and distributed management of global transactions.

Our approach is based on formal analysis of the four listed properties for a given application. If the application can be shown to satisfy the properties, then the developer is assured that the four goals are met for that application. If the application does not satisfy the properties, then one or more of the goals is not met, and the application may need to be revised.

An important question is how well our model, particularly the verification aspect, scales up to commercial applications. In this paper, we have used `Object Z` as the specification language and have shown how model checking can be used to address – in part, but automatically – whether a specification satisfies the necessary properties. For future work, we believe it would be useful to investigate how well existing theorem provers such as PVS [23] can automatically discharge the necessary proof obligations.

Appendix A. Specification of extended banking example

The specification language `Object Z` [9] is used for formalizing the banking application. `Object Z` is based on the specification language `Z` [25]; it is an extension primarily to include object oriented features and temporal logic. Table 1 lists the `Object Z` notation used in our specifications. Figs. 2–4 shows the `Object Z` specification of the Branch, Head Office and the Banking Enterprise described in Example 5.

Semantic atomicity property

Proof obligation. Any partial semantic history H_p defined over a set of transactions 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

Table 1
Relevant Object Z notations

\mathbb{Z}	Set of integers
\mathbb{N}	Set of natural numbers
$\mathbb{P}A$	Powerset of set A
\emptyset	Empty set
$\downarrow A$	Set of all possible objects from A and any derived class
$x \mapsto y$	Ordered pair (x, y)
$A \mapsto B$	Partial function from A to B
$\text{dom}A$	Domain of relation A
$\text{ran}A$	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
\cong	Schema definition
op	Identifier denoting the operation
$\square P$	P always holds
$\diamond P$	Eventually P holds
$\bigcirc P$	In the next state P holds

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 on the execution of *GTrans1Comp*; the precondition of *GTrans1Comp* that checks for the validity of the account is also guaranteed to be satisfied (because the account is the one involved in *GTransfer1* and so it is a valid account). Thus *GTrans1Comp* 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 *GDep1Comp*; *GDep1Comp* can execute if its preconditions are satisfied. The precondition of *GDep1Comp* may be violated if a *GWithdraw1* executes between *GDeposit1* and *GDep1Comp*. However, in this case *GWith1Comp* can execute and in turn establish the precondition of *GDep1Comp*.

Finally we show how *GAudit* transactions can complete by successfully executing *GAudit2*. The history invariant imposes the restriction that *GAudit2* cannot execute when integrity constraints are violated. Note that if *GTransfer1*, *GDeposit1*, *GWithdraw1* violates the integrity constraints, the corresponding compensating subtransactions can execute and establish the integrity constraints after which *GAudit2* can execute.



Fig. 2. Local transactions and subtransactions of branch.

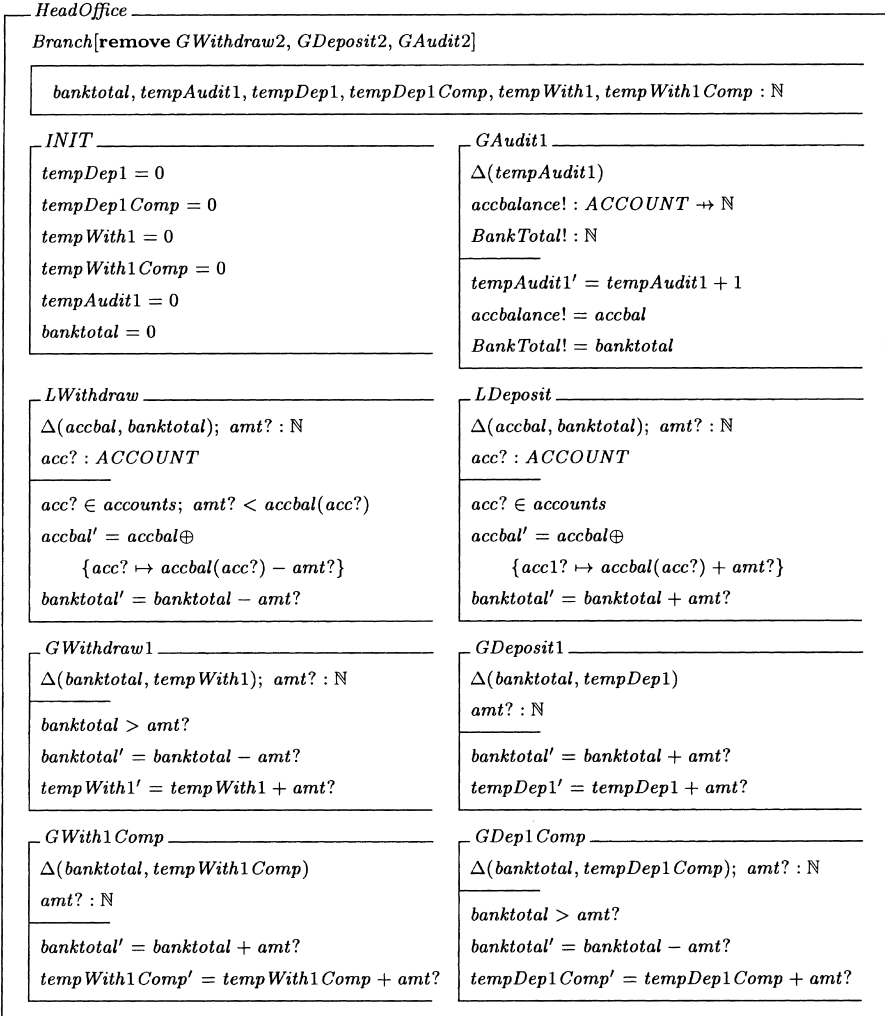


Fig. 3. Local transactions and subtransactions of head office.

Consistent execution property

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

Proof. When a subtransaction of type *GDeposit1* executes, the auxiliary variable $H.tempDep1$ is incremented by the amount of money deposited by the

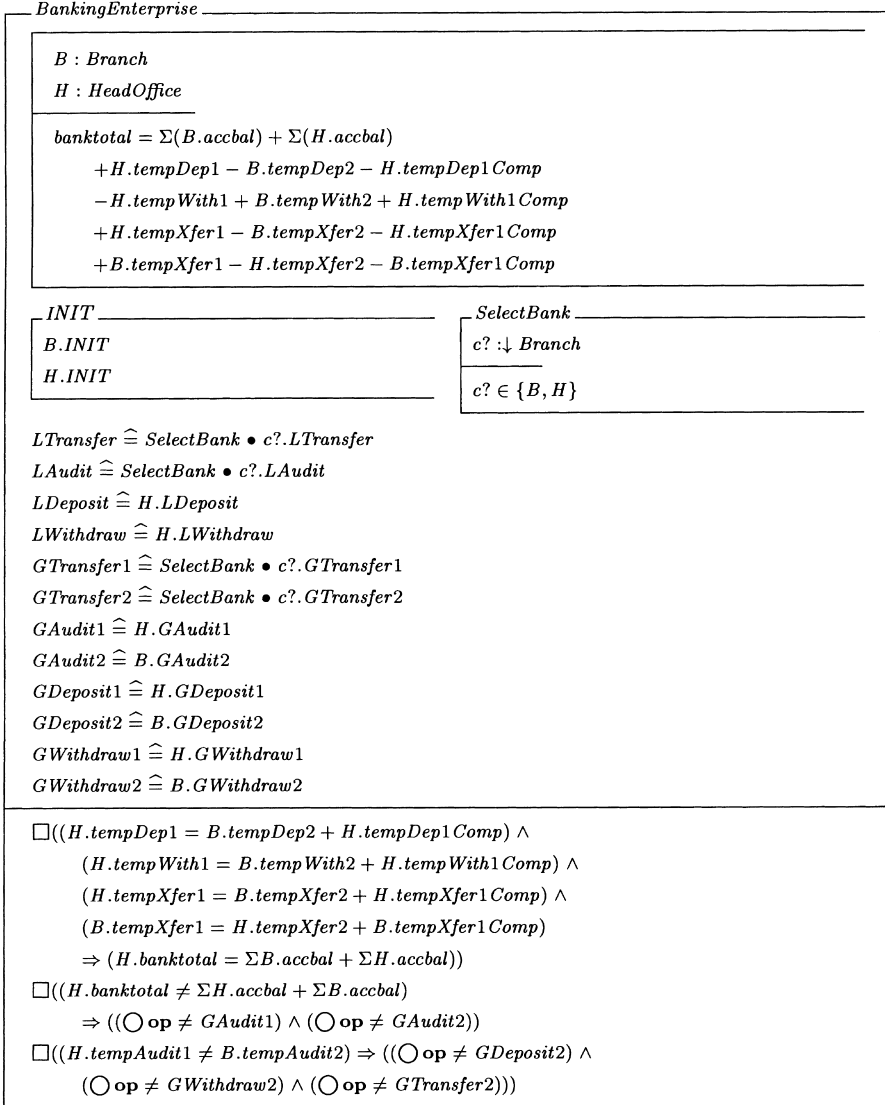


Fig. 4. The banking enterprise.

subtransaction. When the corresponding subtransaction of type $GDeposit2$ executes, the auxiliary variable $B.\text{tempDep2}$ is incremented by the same amount. Or if the corresponding compensating subtransaction of type $GDep1Comp$ executes, the auxiliary variable $H.\text{tempDep1Comp}$ 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 $GDeposit$ subtransactions 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.

Sensitive transaction isolation property

The sensitive transaction isolation property is developed by construction; see Section 4.3 for details.

Appendix B. Refinement of extended banking example

Valid successor set property

Proof obligations.

- (i) Any history generated by a successor set description is equivalent to some correct history generated by the specification consisting of history invariants.
- (ii) The refined specification satisfies the semantic atomicity property.

Proof of (i)

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 subtransactions are committed in H and others whose first subtransactions are committed in B . There can be two types of interleavings. Case (a): The head office has committed one or more subtransactions of type $GAudit1$ and the branch has committed one or more subtransactions of type $GTransfer1$. In such a case, the only way the successor set description allows the history to be completed is the branch executing compensating subtransactions of type $GTrans1Comp$ for all incomplete $GTransfer$ transactions. When all $GTransfer$ transactions have completed, the subtransactions of type $GAudit2$ 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 subtransactions which are not of type *GAudit1* and the branch has committed one or more subtransactions of type *GTransfer1*. In this case all possible interleavings do not involve subtransactions 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 subtransactions are committed in *B* or *H* can be argued similarly.

Proof of (ii)

Consider a 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 partial semantic history is a prefix of the 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 *GTransfer1* has committed at the branch. Since *GTrans1Comp* is in the successor set of *GTransfer1* and the precondition of *GTrans1Comp* (which checks for the validity of the account) is always satisfied, *GTrans1Comp* can always execute and complete *GTransfer*. Next consider a *GTransfer* transaction whose *GTransfer1* subtransaction has committed at the head office. Due to successor set constraints, *GTrans1Comp* can execute provided no other subtransactions of type *GAudit1* have executed after *GTransfer1*. Also the successor set description will prevent *GAudit1* from being executed after a *GTransfer1*. This means that *GTrans1Comp* 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 subtransaction of type *GAudit2* 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 *GTrans1Comp* after which *GAudit2* can execute.

References

- [1] J.M. Atlee, J.D. Gannon, State-based model checking of event driven systems requirements, *IEEE Transactions on Software Engineering* 19 (1) (1993) 13–23.
- [2] P. Ammann, S. Jajodia, I. Ray, Applying formal methods to semantic-based decomposition of transactions, *ACM Transactions on Database Systems* 22 (2) (1997) 215–254.
- [3] O.A. Bukhres, A.K. Elmagarmid, *Object-Oriented Multidatabase Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1996.
- [4] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, A. Silberschatz. On rigorous transaction scheduling, *IEEE Transactions on Software Engineering* 17 (9) (1991) 954–960.

- [5] P.A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987.
- [6] R.K. Batra, M. Rusinkiewicz, D. Georgakopoulos, A decentralized deadlock-free concurrency control method for multidatabase transactions, in: *Proceedings of the International Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992, pp. 72–79.
- [7] Y. Breitbart, A. Silberschatz, Multidatabase update issues, in: *Proceedings of ACM-SIGMOD International Conference on Management of Data*, June 1988, pp.135–142.
- [8] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J.D. Reese, Model checking large software specifications, *IEEE Transactions on Software Engineering* 24 (7) (1998) 498–520.
- [9] D. Duke, R. Duke, Towards a semantics for Object Z, in: D. Bjorner, C.A.R. Hoare, H. Langmaack (Eds.), *VDM'90: VDM and Z*, vol. 428 of *Lecture Notes in Computer Science*, Springer, Berlin, 1990, pp. 242–262.
- [10] W. Du, A.K. Elmagarmid, Quasi serializability: a correctness criterion for global concurrency control in interbase, in: *Proceedings of the International Conference on Very Large Databases*, Amsterdam, Netherlands, 1989, pp. 347–355.
- [11] D. Georgakopoulos, Multidatabase recoverability and recovery systems, in: *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, April 1991, pp. 348–355.
- [12] H. Garcia-Molina, Using semantic knowledge for transaction processing in a distributed database, *ACM Transactions on Database Systems* 8 (2) (1983) 186–213.
- [13] H. Garcia-Molina, Global consistency constraints considered harmful for heterogeneous database systems, in: *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, April 1991, pp. 248–250.
- [14] H. Garcia-Molina, K. Salem, Sagas, in: *Proceedings of ACM-SIGMOD International Conference on Management of Data*, San Francisco, CA, 1987, pp. 249–259.
- [15] D. Jackson, Niptick: a checkable specification language, in: *Proceedings of the Workshop on Formal Methods in Software Practice*, San Diego, CA, January 1996, pp. 60–69.
- [16] C. Morpain, M. Cart, J. Ferrie, J.F. Pons, Maintaining database consistency in presence of value dependencies in multidatabase systems, in: *ACM SIGMOD Record*, Montreal, Canada, June 1996, pp. 459–468.
- [17] K.L. McMillan, *Symbolic model checking: an approach to the state explosion problem*, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, 1992.
- [18] S. Mehrotra, R. Rastogi, Y. Breitbart, H.F. Korth, A. Silberschatz, The concurrency control problem in multidatabases: characteristics and solutions, in: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, CA, 1992, pp. 288–297.
- [19] S. Mehrotra, R. Rastogi, H.F. Korth, A. Silberschatz, Non-serializable executions in heterogeneous distributed database systems, in: *Proceedings of the International Conference on Parallel and Distributed Systems*, Miami Beach, FL, December 1991, pp. 245–252.
- [20] S. Mehrotra, R. Rastogi, H. Korth, A. Silberschatz, A transaction model for multidatabase systems, in: *Proceedings of the International Conference on Distributed Computing Systems* pages, 1992, pp. 56–63.
- [21] M.H. Nodine, S.B. Zdonik, The impact of transaction management on object-oriented multidatabase views, in: O.A. Bukhres, A.K. Elmagarmid (Eds.), *Object-Oriented Multidatabase Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1996, pp. 57–104 (Chapter 3).
- [22] S. Owicki, D. Gries, Verifying properties of parallel programs: an axiomatic approach, *Communications of the ACM* 19 (5) (1976) 279–285.
- [23] S. Owre, J.M. Rushby, N. Shankar, PVS: a prototype verification system, in: *Proceedings of the International Conference on Automated Deduction*, Saratoga, NY, 1992, pp. 748–752.

- [24] R. Rastogi, H.F. Korth, A. Silberschatz, Exploiting transaction semantics in multidatabase systems, in: *Proceedings of the International Conference on Distributed Computing Systems*, Vancouver, Canada, June 1995, pp. 101–109.
- [25] J.M. Spivey, *The Z Notation: A Reference Manual*, second ed., Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [26] J.M. Wing, M. Vazari-Farahani, A case study in model checking software systems, *Science of Computer Programming* 28 (1997) 273–299.