Modular Reasoning about Open Systems: A Case Study of Distributed Commit

R. Das

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge MA 02139, U.S.A.

A. Fekete

Department of Computer Science
University of Sydney
Sydney 2006, Australia

Abstract

We show how to reason about distributed database management systems, in which a commit protocol is used to coordinate activity of several resource managers. This is an interesting case study of an open system, in which each component is developed independently to operate with many possible environments. We give specifications for each resource manager, and a specification for the commit protocol, and show that the whole system is correct as long as each component has the properties required of it. We then show how to prove that specific examples have these properties.

1 Introduction

Commercial computing has changed radically in recent years. Instead of *proprietary systems* in which all components are designed specifically for use together, there are *open systems* where components are taken off-the-shelf and combined. A component of an open system needs to work correctly with many different environments. This naturally raises the question of how to specify the requirements on a component. We want to know that the whole system will be correct, provided each component is independently developed to meet its own specification.

In this paper, we examine one type of system in this way. Our focus is on transaction management in distributed database management systems. These systems are built from separate resource managers, each of which maintains a collection of information. Transactions are provided by users, and each can access data in several resource managers. Linking these is a *commit protocol* that ensures that a transaction does not commit at any site unless its effects are installed at every site where it ran, despite system failures ("crashes") that can cause changes to be lost. The commit protocol is provided by a transaction processing monitor that is distributed through the system, but designed independently of the resource managers.

The classic work of Lindsay *et al.* [4] presents an algorithm to provide transparently distributed transactions. However, the algorithm is given as an integrated system. The mechanism of commit processing is generally abstracted from the complete algorithm under the name of the *two-phase* commit protocol. A number of alternative mechanisms have been proposed for use in a transaction processing monitor. For example, Mohan, Lindsay and Obermack [7] show how to modify two-phase commit so that fewer messages and forced log-writes are needed.

Most of the discussions of commit protocols present an abstract specification expressing requirements on a commit protocol. There is a particularly clear account of this in Section 7.3 of *Concurrency Control and Recovery in Database Systems* [1]. In that book five properties (AC1-AC5) are defined. The interface between the commit protocol and the sites is that each site, where transaction T ran, casts a "vote" (either *Yes* or *No*), and the protocol determines whether to commit T or abort it. The essential property is AC3, which says the protocol is required not to commit T except in cases where all votes are *Yes*.

For use in an open system, however, one must also characterize the behavior of the resource managers. In particular, one must give a condition on when a manager should vote *Yes*. This is generally treated very loosely in the database literature. For example, Mohan *et al.* [7] state that a *Yes* vote indicates that the site "is willing to commit the transaction" and has "force-writ[ten] a prepare log record". The concept of willingness is not further explained in their paper. Bernstein *et al.* provide [1] a more detailed discussion, where they state that a site may vote *Yes* as long as "every value read by T at that site was written by a

transaction that committed" and also "all values written by T at that site are in stable storage – the stable database or the log". Unfortunately, one can construct a system in which each Resource Manager follows the usual rules for two-phase locking, and votes according to the principle given by [1], and where the commit protocol satisfies AC1-AC5, and yet where executions exist that violate transaction semantics. The problem is that a site may process a read operation for transaction T, and a later crash might cause the read-lock to be lost (since lock-tables are kept in volatile memory until a transaction prepares to commit). The rules of Bernstein *et al.* [1] make the site "willing to commit" the transaction. Serializability might be violated, as in the following example involving two transactions T and T' at two sites x and y, each containing one data item. The following history can occur: T reads x, x crashes and recovers – removing the read lock held by T, T' writes x, T' writes y, T' commits, T reads y. In this history by the rules of Bernstein *et al.* [1], both x and y can vote *Yes* for committing T. If T commits, serializability is violated.

The importance of proving the correctness of the system (as well as just specifying the components) is shown by incorrect descriptions in several textbooks. For example, the classic text of Ceri and Pelagatti [2] states that "each participant corresponds to a subtransaction which has performed some write action". If the two-phase commit protocol is run in this way (i.e., not including read-only sites among the participants), and each resource manager keeps its lock-tables in volatile store, then serializability (and database integrity) can be violated, due to loss of read-locks during a crash, just as in the example above.

The specification we give uses the same style of voting interaction, but it is based on a quite different idea from the traditional ones. In essence, we regard a *Yes* vote as meaning "the resource manager will not in the future lose information about the transaction, despite any later crash". Because of this, the commit protocol must check that each manager voted *Yes*, and also that earlier activity of the transaction was not lost in a crash before the vote; it uses a *crash-count* for each site to accomplish this. The mechanism of crash-counts is taken from the Argus system [5]; a related mechanism of "low-water marks" was used by Lindsay *et al.* [4]. Because this is a stronger requirement on the commit protocol than the usual one, we are able to use a very simple and permissive specification for the each resource manager.

In this paper, we deal with a distributed system supporting nested transactions. Nested transactions allow the activity of a transaction to be divided among multiple concurrent "subtransactions", which are protected from interfering with one another. Our work is done using I/O automata [6] as a model. The salient features of the formal model are that it allows description of both problem statements and implemented systems; that the aspect of any entity that is regarded as significant is the set of possible behaviors, that is, sequences of interactions (actions) between the entity and the environment; that it allows one to use state to generate the behaviors required; and that it has theorems that support modular (piece-by-piece) and hierarchical (step-by-step) examination of a complex system. The model includes coverage of liveness issues, but only safety conditions are considered here.

In this paper we first describe the architecture of the system. There are transaction automata, representing the code supplied by the users. There are crashing object automata, representing resource managers. There are local managers, that collectively (together with the communication medium) provide a transaction processing monitor. In Section 2 we list the actions that are in the interface of each component. We also state the top-level requirement, that expresses that the whole system acts correctly. Then in Section 3 we give specifications for the separate components. For a resource manager, the requirement is a condition on the behavior of the manager, saying that when the sequence is reordered in certain ways, that what results is allowed by the type definition of the resource. For the local managers, we require that their collective behavior is a behavior allowed to a particular automaton (called the atomic commit controller). In Section 4, we show the correctness of any system built from components each with the appropriate property, and also show by example how to verify that particular resource managers or commit protocols have the properties required of them.

2 The system structure

The model links all transactions together into a tree, organized by the parent-child relationship between a caller and the called transaction. For uniformity, we include a root transaction T_0 corresponding to the human users, so that "top-level" transactions appear as children of T_0. For simplicity, we assume that each transaction's name encodes all relevant facts, including its arguments, the code it runs, and the site where it runs. We also include as leaves of the tree, all the accesses to resource managers, each as a child of the transaction that invoked it. If T is an access to the resource X, and v is its return value,

we call the pair (T, v) an operation of X.

Now we can define the system decomposition appropriate for describing a system in which sites can crash, and where some commit protocol is used to check before the commit of a top-level transaction. Specifically, we define a *crash system*, which is composed of *transaction automata*, *crashing object automata*, *local manager automata* and a *communication medium*. We refer to the sequence of actions that can occur in an execution of a crash system as a *crash behavior*.

Transaction Automata. A non-access transaction T is modeled as a *transaction automaton* A_T. The **CREATE** input action "wakes up" the transaction. Each **REQUEST_CREATE** output action is a request by T to create a particular child transaction (including a child access as a special case). Each **REPORT_COMMIT** input action reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. Each **REPORT_ABORT** input action reports to T the unsuccessful completion of one of its children, without returning any other information. The **REQUEST_COMMIT** action is an announcement by T that it has finished its work, and includes a value recording the results of that work. We leave the executions of particular transaction automata largely unconstrained; the choice of which children to create and what value to return will depend on the particular implementation.

Notice that we model separately the parent invocating a child, and the child beginning to run, since in a distributed system there will be a delay while the message is transmitted. For uniformity, we use the same notation (**REQUEST_CREATE** and later **REPORT_COMMIT**) to model the request for an access to perform some action at a resource manager, and the subsequent response, as we use for subtransactions.

Crashing Object Automata. A crashing object automaton C_X represents one resource manager. It encapsulates the data stored for the resource X, including perhaps multiple versions, log entries, locks or timestamps. In the theory of [3], each resource also comes with a serial specification S_X, that represents its type in the absence of concurrency and failure.

The crashing object automaton C_X has an interface through which it receives invocations of accesses (**CREATE**) and returns results (**REQUEST_COMMIT**), and receives information about the fate of transactions (**INFORM_COMMIT** and **INFORM_ABORT**). In this, it is similar to the generic object of [3]. It also has an extra **CRASH** action and two extra classes of actions: **PREPARE_REQUEST** and **VOTE_YES**. The **CRASH** action models the loss of volatile storage at the site where the object resides.[1] Notice that since **CRASH** is an input, it may occur at any time (because of the input-enabled rule of the I/O automaton model). A **PREPARE_REQUEST** action indicates to the object that a top-level transaction (that is, a child of T_0, where T_0 models the user of the system) has finished running, and the object should then save the transaction's results on stable storage. Once this has been done, the **VOTE_YES** action is the object's response, announcing that it will be able to process a later **INFORM_COMMIT** action properly, even if **CRASH** events occur. (We do not include an explicit negative response to a **PREPARE_REQUEST** in our model; rather, if no positive response is given, the transaction can never commit. In practice, a timeout would eventually lead to the transaction aborting.)

Local Manager Automata. We model the activity of the commit protocol at a single site n by a *local manager automaton* L_n, which represents the *transaction manager*. Once a top-level transaction requests to commit, it is the local manager that issues any **PREPARE_REQUEST** action to a resource manager located at the site, and that reacts to the **VOTE_YES** action in response. Different local managers interact with one another by passing messages: the **SEND** actions and **RECV** actions model the sending and receipt of messages, respectively. For simplicity, we use the local managers for all communication[2] between sites. For example, the local manager also accepts a **REQUEST_CREATE** action from a transaction at the site, and sends a message to its peer at the site of the requested child. On receiving this information the local manager issues the **CREATE** action to the child transaction (or resource manager, in case the child is an access). Similarly the local manager accepts a **REQUEST_COMMIT** input from a transaction or resource manager, acts to complete the transaction (either **COMMIT** or **ABORT**), and then sends information to the local manager at the site where the parent is running, after which that manager issues the **REPORT_COMMIT** or **REPORT_ABORT** action as appropriate. The information about completion is also sent to local managers at other sites, so that resource managers can be given information in **INFORM_COMMIT** or **INFORM_ABORT** actions. We also assume that the local manager can be affected

[1] When specific algorithms are discussed, as in section 4, the **CRASH** will have the effect of causing the state to change to some value that depends only on the previous value of the stable components.

[2] In commercial systems, it is more common for the resource managers to communicate directly, passing to the commit protocol indications of what is being done.

by the crash of the site.

The Communication Medium. The communication network is also modelled, as an automaton. The interface of the communication network automaton is as follows. The input actions are **SEND**(m)**AT**(n)**TO**(p), and the output actions are **RECV**(m)**AT**(n)**FROM**(p).

Requirements for the Crash System. Next, we state a specification for the complete distributed database management system, representing a closed universe containing users and user-supplied code (transaction automata) resource managers (crash object automata), peers forming the commit protocol (local manager automata), and an underlying message passing communication medium. The key property required of a distributed database management system is *transparency*, that is, the system should not be functionally distinguishable from a single-site database management system (DBMS). Since any single-site DBMS supports transactions, the distributed system must do so too. We use the approach of Fekete *et al.*. Precise details of the definition can be found in [3], but in essense, we define an ideal system, called a *serial system*, in which the same transaction automata are present, but the execution is controlled so that sibling transactions are run without concurrency, no transaction fails after taking some steps, and each resource is implemented by its abstract type without concurrency control or recovery mechanisms. We say that a particular execution α of a DBMS is *serially correct for* T_0 provided there exists an execution α' of the serial system such that the activity of T_0 (representing the users) is the same in the two sequences. We will use exactly the same definition as the requirement on a crash system: every execution of the crash system must be serially correct for T_0.

3 Specifications of the components

Requirements on Transaction Automata. We make only minor restrictions on the construction of transaction automata, since these represent code provided by the user. We require that each automaton *preserve transaction well-formedness*. A transaction well-formed sequence is always a prefix of a sequence that starts with **CREATE**(T), ends with **REQUEST_COMMIT**(T,v), and in between has some interleaving of a collection of two-element sequences **REQUEST_CREATE**(T')**REPORT_COMMIT**(T',v'), for various children T' of T. Thus each transaction automaton must not issue any output that violates this pattern, unless it had already been violated by an earlier input. Notice that we do not restrict the transaction's choice of which children to request, nor its choice of return value.

Requirements on Crash Object Automata. We now formally define the concept of *local crash atomicity*, which is the obligation we place on each resource manager. In essence, as in dynamic atomicity in [3], the obligation is that in each crash behavior, the values returned to accesses by the crash object automaton must be such that the serial specification of the resource is allowed to act in the way described by rearranging these accesses into any order that might have occurred in a serial execution if siblings are run in the order given by their time of completion in real (concurrent) system.

We first collect some elementary properties of a crashing object under the term *crashing object well-formedness*. A sequence β of actions π is said to be *crashing object well-formed* for X provided that all the following conditions hold: there is at most one **CREATE**(T) event in β for any access T; there is at most one **REQUEST_COMMIT** event in β for any access T; if there is a **REQUEST_COMMIT** event for T in β, then there is a preceding **CREATE**(T) event in β; if there is a **VOTE_YES** event for T in β, then there is a preceding **PREPARE_REQUEST** event for T in β; if an **INFORM_COMMIT_AT**(X)**OF**(T) event occurs in β and T is an access to X, then there is a preceding **REQUEST_COMMIT** event for T; there is no transaction T for which both an **INFORM_COMMIT** event and an **INFORM_ABORT** event at X for T occur in β. We require that the crashing object automaton preserve crashing object well-formedness.

We define *local visibility*: we say that T is locally visible to T' in a sequence β of actions of a crashing object C_X if β contains **INFORM_COMMIT_AT**(X)**OF**(U) for every U in $ancestors(T) - ancestors(T')$. We also define *local-completion*(β) to be the binary relation on accesses to X where $(U, U') \in$ *local-completion*(β) if and only if $U \neq U'$, β contains **REQUEST_COMMIT** events for both U and U', and U is locally visible at X to U' in β', where β' is the longest prefix of β not containing the given **REQUEST_COMMIT** event for U'. Define a sequence ξ of operations of X to be *transaction-respecting* provided that for every transaction name T, all the operations for descendants of T appear consecutively in ξ.

Suppose that β is a finite crash object well-formed sequence of external actions of C_X. Then *local-views*(β) is the set of sequences defined as follows. Let Z be the set of all operations (T, v), such that **REQUEST_COMMIT**(T,v) occurs in β and T is

locally visible at X to T_0 in β. Then the elements of *local-views*(β) are all the sequences that can be formed by reordering Z according to a transaction-respecting total ordering consistent with the partial order *local-completion*(β) on the transaction components, and then replacing each operation (T, v) by **CREATE**(T)**REQUEST_COMMIT**(T,v). We say a sequence β of actions of C_X is *autonomy-respecting* for X provided that the following holds for every T that is an access to X: if β contains a **REQUEST_COMMIT** event ϕ for T and T is locally visible to T_0 in β, then there is a **VOTE_YES_AT**(X)**FOR**(U) event π that follows ϕ in β, and furthermore no **CRASH** event occurs between ϕ and π in β, where U is the unique transaction such that T is a descendant of U and *parent*$(U) = T_0$.

We can now combine all the above definitions, to express formally the obligation placed on each resource manager. We say that crashing object automaton C_X for object name X is *locally crash atomic* if whenever β is a finite crashing object well-formed behavior of C_X that is autonomy-respecting then every sequence in *local-views*(β) is a finite behavior of S_X.

Requirements on Local Manager Automata. We express our requirements on the local managers in a very different way. Rather than specify one local manager, we make a condition on the composition of all the local managers together with the communication medium. To express this condition, we define an explicit global automaton called *the atomic commit controller*, that acts as a specification of the commit protocol service as a whole. We then require that whenever α is a sequence of actions that is a transaction well-formed and crashing-object well-formed behavior of the composition of local managers and communication medium, then α is also a behavior of the atomic commit controller.

There is a single atomic commit controller. It is the maximally non-deterministic global automaton that acts as required for a transaction processing monitor, using the semantics described in the introduction. Fundamentally, it acts as a communication medium. For example, it receives **REQUEST_COMMIT**(T,v) from T, reaches an internal decision point represented as **COMMIT**(T), and eventually gives **REPORT_COMMIT**(T,v) to T's parent, and also passes **INFORM_COMMIT_AT**(T)**OF**(X) to C_X. In this, it behaves in much the same way as the generic controller in Fekete *et al.* [3]. The main additional feature is that, after a top-level transaction T has requested to commit but before the commit occurs, the controller asks objects to *prepare* to commit the transaction, that is, to store its effects so that future crashes will not destroy the information. Each object involved issues a *vote* when this has been done, and the transaction can commit only when a vote has been received from every object at which a (non-orphan) descendant access has run. The controller also checks that the descendant accesses at any site all ran within the same *incarnation* of that site as the **VOTE_YES** action.

Each state s of the atomic commit controller consists of the following components: $s.create\_requested$, $s.created$, $s.commit\_requested$, $s.committed$, $s.aborted$ and $s.reported$, $s.crashcount$, $s.crashrec$, and $s.ready$. The first six are sets of transactions or operations, and they simply record the actions that have happened; they are already present in the generic controller of [3]. The component $s.crashcount$ is a function from site names to non-negative integers. For any site n, $crashcount(n)$ is initially zero. In any state it represents the number of crashes that have occurred at the site n. The component $s.ready$ is a function from top-level transaction names (those which are children of T_0) to sets of objects. Initially $s.ready(T)$ is the empty set. In any state it represents the objects that are able to commit T. The component $s.crashrec$ is a partial function from access names to non-negative integers. Initially $s.crashrec(T)$ is undefined, and in any state it represents the number of crashes of the site of T that occurred before the **REQUEST_COMMIT** for T.

We introduce derived variables, writing $s.completed = s.committed \cup s.aborted$, and, for each T, writing $s.visible(T)$ for the set of transaction names T' such that every element of $ancestors(T') - ancestors(T)$ is in $s.committed$; we also define $s.included(T, X)$ to be $s.visible(T) \cap descendants(T) \cap accesses(X)$; we define $s.involved(T)$ to be the set of object names X such that $s.included(T, X) \neq \emptyset$.

Figure 1 shows the code of the atomic commit controller. The transitions for **REQUEST_CREATE**, **CREATE**, **ABORT**, **REPORT_COMMIT**, **REPORT_ABORT**, **INFORM_COMMIT** and **INFORM_ABORT** actions, and also for **COMMIT** transitions for transactions except those whose parent is T_0, and **REQUEST_COMMIT** actions for transactions that are not accesses, are all straightforward (and identical to those in [3]). They simply record requests in appropriate variables, and deliver them later. The transition for the **CRASH** action simply increases the appropriate *crashcount*. The transition for a **VOTE_YES** action records the site in the *ready* set, provided the *crashcount* indicates that the vote occurs when no preceding **CRASH** could have destroyed needed information. Notice how the seman-

tic requirement on the commit protocol is expressed in the precondition of the COMMIT of a top-level transaction, which checks that each participant (that is, each site in *involved*(T)) has voted at an appropriate point (as indicated by being in *ready*(T)).

4 Verification

Having proposed a collection of requirements for the components of a crash system, we still need to prove the correctness (according to the requirements for the whole system) of a crash system in which each component meets the appropriate condition. We will also need ways to prove that specific components do meet their separate requirements.

Correctness of the Component Specifications. The value of our collection of requirements on components is expressed in the following Theorem.

Theorem 4.1 *Suppose that each transaction automaton preserves transaction well-formedness, that each crashing object automaton preserves crashing object well-formedness and is local crash atomic, and that every well-formed behavior of the composition of all the local managers with the communication medium is also a behavior of the atomic commit controller. Then each behavior of the system is serially correct for* T_0.

This result can be proved by using Proposition 46 of [3], which expresses the fundamental intuition that the users see satisfactory behavior so long as each object returns values in such a way that when serialized in the completion order, the activity is allowed by the serial specification. Once the definitions are all unwound, the constraint on the local managers imply that the behavior β of each crashing object is autonomy-respecting, and that the serialization in completion order is one of the sequences in the set *local-views*(β). The local crash atomicity of the crashing object now implies that this sequence is a behavior of the serial specification, exactly as needed to apply Proposition 46 of [3].

A Particular Commit Protocol. Here we demonstrate the modularity afforded by our system decomposition, and present one way to construct local managers that collectively provide the functionality of the atomic commit controller. The algorithm used is based on the standard two-phase commit protocol. Our presentation retains a lot of generality through nondeterminism. For example, we allow a large choice in which information to send in a message, and in which destinations it is sent to. More detailed algorithms may make specific restrictions, sending information only when needed, and only to sites that need to know. The correctness of such a detailed algorithm will follow from the correctness of this one, since the detailed algorithm will have as its behaviors a subset of the behaviors of the non-deterministic algorithm we present.

The local manager L_n at site n has the interface described above. The state components of state s of the local manager L_n include all those that are components of a state of the atomic commit controller, with the same types and initial values. Each component represents what is known at the site n about transaction status, etc., corresponding to the variable of the same name in the atomic commit controller.[3] There is also an additional state component *s.inc*. For each transaction name T and object name X, *s.inc*(T, X) is a set of accesses to X. Initially *s.inc*(T, X) is $\{T\}$ if T is an access, and otherwise it is empty. The component *s.inc* reflects local knowledge corresponding to the derived variable *included* in the atomic commit controller.

We use three derived state components: *s.completed* = *s.committed* $\cup$ *s.aborted*; for each T, *s.visible*(T) is the set of transaction names T' such that every element of *ancestors*(T') $-$*ancestors*(T) is in *s.committed*; for each top-level T, *s.involved*(T) is the set of object names X such that *s.inc*$(T, X) \neq \emptyset$.

In our system, we will use the following format for the messages sent between local managers: a *message* is a record whose components are those of a state of a local manager except for *crashcount*. Thus a message indicating that T has committed could be modeled as a record m in which m.*committed* $= \{T\}$, and the other components are empty. By using this general notation, we allow ourselves to model piggybacking multiple information into a single message, and also make the code easy to write.

The transition relation of a local manager automaton is given in Figure 2.

At this point, a few words must be mentioned about the SEND output action of the local manager automata. As can be seen from the transition relation, the effect of the SEND(m)AT(n)TO(p) action is the placing of a subset of the total state information of a local manager in the message. However, we have to ensure that the "right" information reaches the "right" site at the "right" time. This is important, for example, when determining the list of participants in a commit decision. Hence, we adopt a piggybacking strategy which gives such a guarantee. For example,

[3] For uniformity, we retain *crashcount* as a function from sites to integers, though in fact we ignore all the values recorded for sites other than n itself.

we piggyback the *inc* information for a transaction T with any message from T's site recording T as in $m.committed$, ensuring that both the *inc* information for T and the *committed* information reach any destination site at the same time.

The communication network we consider is neither lossless nor order-preserving, but it does not allow corruption or duplication of submitted messages. It is modelled as an automaton. Each state s of the communication network automaton consist of just one component: $s.messages$. Thus, each state is just a set of messages. The input action **SEND**(m)**AT**(n)**TO**(p) adds a message, whose destination is node p, to the set, and the output action **RECV**(m)**AT**(n)**FROM**(p) delivers a message to p only if it is in the set, and deletes the message from the collection of messages, preventing multiple deliveries.

Correctness of the Commit Protocol. The correctness of the commit protocol described above is expressed formally by saying that for any behavior of the system ("the distributed commit controller") that is the composition of all local manager automata together with the communication network automaton, the subsequence of that behavior (consisting of actions in the interface of the atomic crash controller) is a behavior of the atomic commit controller. We remark that this subsequence is exactly the subsequence formed by hiding all **SEND** and **RECV** actions.

The correctness is shown with the help of a possibilities mapping [6] relating states of the distributed commit controller and states of the atomic commit controller. The mapping relates states that represent "essentially" the same information. Let us denote a state of the distributed commit controller by s, which is given by a state for each component: $s[n]$ for the state of the local manager at site n, and $s[c]$ for the state of the communication medium. For brevity, we abuse notation by writing $m \in s[c].messages$ to refer to the set of m for which there exist p and q such that (m,p,q) is in $s[c].messages$. Let us denote a state of the atomic commit controller as t. The possibilities mapping relates s to t provided the following conditions are met:

1. $t.create\_requested = \bigcup_{n \in N} s.[n]create\_requested \cup \bigcup_{m \in s[c].messages} m.create\_requested,$

2. $t.created = \bigcup_{n \in N} s.[n]created \cup \bigcup_{m \in s[c].messages} m.created,$

3. $t.commit\_requested = \bigcup_{n \in N} s.[n]commit\_requested \cup \bigcup_{m \in s[c].messages} m.commit\_requested,$

4. $t.committed = \bigcup_{n \in N} s.[n]committed \cup \bigcup_{m \in s[c].messages} m.committed,$

5. $t.aborted = \bigcup_{n \in N} s.[n]aborted \cup \bigcup_{m \in s[c].messages} m.aborted,$

6. $t.reported = \bigcup_{n \in N} s.[n]reported \cup \bigcup_{m \in s[c].messages} m.reported,$

7. $t.ready(T) = \bigcup_{n \in N} s.[n]ready(T) \cup \bigcup_{m \in s[c].messages} m.ready(T)$ for each T where $parent(T) = T_0$,

8. $t.crashcount(n) = s.[n]crashcount(n)$ for all sites n,

9. $t.crashrec(T) = s.[site(T)]crashrec(T)$ for all accesses T.

In order to prove that this is a possibilities mapping, we need to use some invariants of the system. For example, in any state s of the distributed commit controller, if $T \in s.[n]create\_requested$ or $T \in m.create\_requested$ for some $m \in s[c].messages$, then $T \in s[site(parent(T))].create\_requested$. Also, if $T \in s.[n]created$ or $T \in m.created$ for some $m \in s[c].messages$, then $T \in s[site(T)].created$. Similar properties hold for the other state components, reflecting that information available anywhere in the system must also be held at the site where the corresponding event occurred.

A Resource Manager. It is quite straightforward to provide a locally crash atomic object, simply by taking any dynamic atomic algorithm (such as those given in [3]) and keeping all its states on stable storage. In this case the object can vote to commit any transaction as soon as the vote is requested by a **PREPARE_REQUEST** action. Of course, if this were implemented in a real system, the performance would be terrible, since each action requires a write to stable storage, which is generally much slower than a write to volatile storage. Here we present an algorithm that provides local crash atomicity, and yet writes to stable storage only when transactions vote, rather than at every access.

The algorithm we give is based closely on Moss's algorithm for read-update locking, as presented using

I/O automata in [3]. In fact, during normal processing, the algorithm is identical (operating entirely on volatile state). The **VOTE_YES** action for any transaction involves copying the current volatile state into stable storage, and the effect of a **CRASH** is to destroy the existing volatile state, and cause it to be replaced[4] by the copy from stable storage (as recorded at the most recent **VOTE_YES**). It is easy to write a transition relation for an automaton CV_X, expressing these ideas.

Correctness of the Resource Manager. The correctness of the algorithm described by CV_X is formally expressed as the statement that CV_X is locally crash atomic. To prove this statement, we consider any behavior β of CV_X. One can form a sequence $clean(\beta)$ as follows: delete any action in β which is followed by a **CRASH** without an intervening **VOTE_YES** (these are the actions whose effects are lost due to crashes), also delete all **PREPARE_REQUEST**, **VOTE_YES**, and **CRASH** actions, and finally append to the sequence a copy of every **INFORM_COMMIT** action in β. The relationship between the code of CV_X and that of M_X implies that $clean(\beta)$ is a behavior of M_X. Furthermore, if β is autonomy-respecting then the set of events in β that are locally visible to T_0 in β is exactly the same as the set of events that occur in $clean(\beta)$ and are locally visible to T_0 in $clean(\beta)$. Also, the local-completion order in $clean(\beta)$ is a subrelation of the local-completion order in β. Thus each sequence that is in $local\text{-}views(\beta)$ is a reordering, consistent with the local-completion order, of the events in $clean(\beta)$ that are visible to T_0. The paper [3] shows M_X to be locally dynamic atomic, which means that each of these sequences is a behavior of S_X. Since this holds for an arbitrary autonomy-respecting behavior of CV_X, this is exactly what is needed to show that CV_X is locally crash atomic.

In fact, one can modify the algorithm and improve the performance markedly, by writing far less to stable storage during a **VOTE_YES**. In fact, all that needs to be saved (and later restored) are the locks and versions held for transactions that are either T_0 itself, or children of T_0.

5 Conclusion and further work

We have proposed a specification of transaction management, identifying requirements on the commit protocol and also on the conditions under which sites may vote to commit. We have shown that transactional semantics are produced by systems that fit this framework, and also how one can then present specific protocols with the required properties.

There is much that can still be done. The distributed commit protocol we have presented is not itself efficiently fault-tolerant; that is, the **CRASH** action does not damage the information used by the local manager itself, so all of this must be kept on stable storage. In fact, the usual two-phase commit protocols are designed to tolerate failures of this sort, while not keeping much information in stable storage. It would be good to adapt these algorithms to maintain crash-counts, and then verify that the resulting systems do compose to implement the atomic crash controller.

It would also be interesting to model and verify other specifications for commit protocols, such as those corresponding to the traditional specification in which the commit protocol need not maintain crash-counts.

Acknowledgements. We thank Nancy Lynch, William Weihl and Michael Merritt for many useful comments on this material. The presentation was greatly helped by discussions with Jim Burns and Betty Salzberg.

[4] Recall that we have included the complete post-crash restart process in our model as part of the **CRASH** action itself.

References

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

[2] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, 1984.

[3] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. *Journal of Computer and System Sciences*, 41(1):65–156, 1990.

[4] B. Lindsay, P. Selinger, C. Galtieri, J. Gray, R. Lorie, T. Price, F. Putzolu, I. Traiger, and B. Wade. Notes on distributed databases. Technical Report RJ2571(33471)7/14/79, I.B.M. San Jose Research Laboratory, July 1979.

[5] B. Liskov. Distributed computing in Argus. *Communications of ACM*, 31(3):300–312, March 1988.

[6] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[7] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *Transactions on Database Systems*, 11(4):378–396, December 1986.

REQUEST_CREATE(T)
Effect:
$s.create\_requested = s'.create\_requested \cup \{T\}$

REQUEST_COMMIT(T,v)
Effect:
$s.commit\_requested = s'.commit\_requested \cup \{(T, v)\}$

CREATE(T)
Precondition:
$T \in s'.create\_requested - s'.created$
Effect:
$s.created = s'.created \cup \{T\}$

COMMIT(T), where $parent(T) \neq T_0$
Precondition:
$(T, v) \in s'.commit\_requested$
$T \notin s'.completed$
Effect:
$s.committed = s'.committed \cup \{T\}$

ABORT(T)
Precondition:
$T \in s'.create\_requested - s'.completed$
Effect:
$s.aborted = s'.aborted \cup \{T\}$

VOTE_YES_AT(X)FOR(T)
Effect:
$s.ready(T) = s'.ready(T) \cup \{X\}$
if $s'.crashrec(T') = s'.crashcount(site(X))$
for every $T' \in s'.included(T, X)$
$s.ready(T) = s'.ready(T)$ otherwise

CRASH(n)
Effect:
$s.crashcount(n) = s'.crashcount(n) + 1$

PREPARE_REQUEST_AT(T)FOR(X)
Precondition:
$(T, v) \in s'.commit\_requested$ for some v

REPORT_COMMIT(T,v)
Precondition:
$T \in s'.committed$
$(T, v) \in s'.commit\_requested$
$T \notin s'.reported$
Effect:
$s.reported = s'.reported \cup \{T\}$

REPORT_ABORT(T)
Precondition:
$T \in s'.aborted$
$T \notin s'.reported$
Effect:
$s.reported = s'.reported \cup \{T\}$

INFORM_COMMIT_AT(X)OF(T)
Precondition:
$T \in s'.committed$

INFORM_ABORT_AT(X)OF(T)
Precondition:
$T \in s'.aborted$

REQUEST_COMMIT(T,v), T an access
Effect:
$s.commit\text{-}requested = s'.commit\text{-}requested \cup \{(T, v)\}$
$s.crashrec(T) = s'.crashcount(n)$
where $n = site(T)$

COMMIT(T), $parent(T) = T_0$
Precondition:
$(T, v) \in s'.commit\_requested$ for some v
$T \notin s'.completed$
$s'.involved(T) \subseteq s'.ready(T)$
Effect:
$s.committed = s'.committed \cup \{T\}$

Figure 1: Transition Relation for the Atomic Commit Controller

REQUEST_CREATE(T)
Effect:
$s.create\_requested = s'.create\_requested \cup \{T\}$

REQUEST_COMMIT(T,v)
Effect:
$s.commit\_requested = s'.commit\_requested \cup \{(T, v)\}$
$s.inc(T, X) = \bigcup_{T' \in children(T) \cap s'.committed} s'.inc(T', X)$
for each X
$s.crashrec(T) = s'.crashcount(n)$ if T is an access

VOTE_YES_AT(X)FOR(T)
Effect:
$s.ready(T) = s'.ready(T) \cup \{X\}$
if $s'.crashrec(T') = s'.crashcount(n)$
for each $T' \in s'.inc(T, X)$,
$s.ready(T) = s'.ready(T)$ otherwise

RECV(m)AT(n)FROM(p)
Effect:
$s.create\_requested = s'.create\_requested \cup$
$m.create\_requested$
$s.created = s'.created \cup m.created$
$s.commit\_requested = s'.commit\_requested \cup$
$m.commit\_requested$
$s.committed = s'.committed \cup m.committed$
$s.aborted = s'.aborted \cup m.aborted$
$s.reported = s'.reported \cup m.reported$
$s.ready(T) = s'.ready(T) \cup m.ready(T)$
for each top-level T
$s.inc(T, X) = s'.inc(T, X) \cup m.inc(T, X)$
for each T and each X
$s.crashrec = s'.crashrec \cup m.crashrec$

COMMIT(T), (where $parent(T) \neq T_0$)
Precondition:
$(T, v) \in s'.commit\_requested$ for some v
$T \notin s'.completed$
Effect:
$s.committed = s'.committed \cup \{T\}$

REPORT_COMMIT(T,v)
Precondition:
$T \in s'.committed$
$(T, v) \in s'.commit\_requested$ for some v
$T \notin s'.reported$
Effect:
$s.reported = s'.reported \cup \{T\}$

REPORT_ABORT(T)
Precondition:
$T \in s'.aborted$
$T \notin s'.reported$
Effect:
$s.reported = s'.reported \cup \{T\}$

CRASH(n)
Effect:
$s.crashcount(n) = s'.crashcount(n) + 1$

PREPARE_REQUEST_AT(X)FOR(T)
Precondition:
$(T, v) \in s'.commit\_requested$ for some v

CREATE(T)
Precondition:
$T \in s'.create\_requested - s'.created$
Effect:
$s.created = s'.created \cup \{T\}$

COMMIT(T), (where $parent(T) = T_0$)
Precondition:
$(T, v) \in s'.commit\_requested$ for some v
$T \notin s'.completed$
$s'.involved(T) \subseteq s'.ready(T)$
Effect:
$s.committed = s'.committed \cup \{T\}$

ABORT(T)
Precondition:
$T \in s'.create\_requested - s'.completed$
Effect:
$s.aborted = s'.aborted \cup \{T\}$

INFORM_COMMIT_AT(X)OF(T)
Precondition:
$T \in s'.committed$

INFORM_ABORT_AT(X)OF(T)
Precondition:
$T \in s'.aborted$

SEND(m)AT(n)TO(p)
Precondition:
$m.create\_requested \subseteq s'.create\_requested$
$m.created \subseteq s'.created$
$m.commit\_requested \subseteq s'.commit\_requested$
$m.committed \subseteq s'.committed$
$m.aborted \subseteq s'.aborted$
$m.reported \subseteq s'.reported$
$m.ready(T) \subseteq s'.ready(T)$
for each top-level T
$m.inc(T, X) \subseteq s'.inc(T, X)$
for each T and each X
if $((T, v) \in m.commit\_requested$ for some v,
$T \notin m.completed$, $parent(T) = T_0$,
and $site(T) = n$),
or ($T \in m.committed$ and $site(T) = n$), then
$m.inc(T, X) = s'.inc(T, X)$ for each X
$m.crashrec \subseteq s'.crashrec$
if $T \in m.inc(T', X)$ for any T', X, then
$m.crashrec(T) = s'.crashrec(T)$

Figure 2: Local Manager Transition Relation