Internal Consistency of A Distributed Transaction System

with Orphan Detection

by

John A. Goree, Jr.

S.B. Massachusetts Institute of Technology
(1981)

Submitted to the Department of Electrical Engineering
and Computer Science in partial fullfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

at the

MASSACHUSETTS INSITUTE OF TECHNOLOGY

January, 1983

Signature of Author______________________________
Department of Electrical Engineering and Computer Science
January 4, 1983

Certified by______________________________
Prof. Nancy A. Lynch, Thesis Supervisor

Accepted by______________________________
Prof. Arthur C. Smith, Chairman., E.E.C.S. Department Committee on Graduate Students

Internal Consistency of a Distributed Transaction System with Orphan Detection

by

John A. Goree

Submitted to the Department of Electrical Engineering and Computer Science on January 4, 1983 in partial fulfillment of the requirements for the degree of Master of Science

Abstract

This thesis defines a property called "view-serializability," which formalizes internal consistency for a system of nested atomic transactions. Internal consistency is a stronger condition than the usual notion of database consistency, because it takes into account the views of transactions which will never commit. In a distributed system, local aborts of remote subactions and crashes of nodes can generate *orphans*: active actions which are descendants of actions that have aborted or are guaranteed to abort. Because it is not always feasible or efficient to elimate orphans immediately, special care is needed to insure that they see consistent system states when they are allowed to continue running. We investigate a particular dynamic detection strategy designed to detect orphans before they violate internal consistency. This algorithm piggybacks abort and crash information on the normal messages between nodes. We consider a simpler algorithm that only handles orphans arising from explicit aborts. We describe the simplified orphan detection algorithm at various levels of abstraction, using an algebraic model convenient for describing asynchronous systems. The highest-level model is specified in terms of a (virtual) global state. At this level of abstraction we require that the states generated by the model satisfy view-serializability. Lower-level models progressively localize the description of the algorithm's operation, and the lowest level of abstraction presents a fully distributed model of the (simplified) orphan detection scheme.

Keywords: concurrency control, orphans, transaction, serializability, internal consistency.

Thesis Supervisor: Nancy A. Lynch
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisor, Professor Nancy Lynch, for the personal attention she has given to this project. Nancy's insistence on simplicity has helped me to make this thesis readable, and her rapid review of my drafts has enabled me to meet a tight schedule. Professor Barbara Liskov has also made many valuable suggestions. Gene Stark and Bill Weihl have contributed several of their own ideas, and they have made many helpful comments. They and many of the other graduate students at the Lab have provided encouragement and have occasionally shared mutual frustration; I thank them and wish them luck in rapidly completing their own theses.

Only an automaton can function without diversions and companionship; I am grateful that several of my friends have prevented me from becoming just another computer at Technology Square. Above all, I thank Gary Fedder, Brian Feldman, Neil Savasta, Wayne Seltzer, and Steve Weiss: without their support I could never have completed my graduate work. I commend all my roommates for their remarkable patience with my moods. Thanks also to Marian Evatt for the sailing lessons, to Mark Radka and Rita Nothaft -- my Peace Corps pals -- for showing me that there is *always* a way out, and to Ruth Fricker and Lora Silverman for frequent doses of good cheer. Finally, a special thanks to the Thursday Night Club for our many stimulating intellectual discussions at the Cask and Flagon.

CONTENTS

1. Introduction

Production of concurrent programs is a much more difficult task than production of sequential programs. The sequential nature of human thought severely limits programmers' ability to manage the complexity of parallel processes. Distributed environments compound these difficulties; robust programs must cope with non-local failures and with incomplete information about the global state of a system. Primitives developed for local, sequential programming have proven inadequate for software development in distributed, concurrent systems. Additional mechanisms have been suggested which allow programmers to think about concurrent programs for distributed systems using largely sequential reasoning.

Current research [Liskov82, Best81] stresses use of the *atomic transaction* as a tool for distributed software. Atomic transactions can insulate users from both the effects of concurrency and the effects of failures, greatly simplifying reasoning about a system. If transactions are truly atomic, then neither users *nor the transactions themselves* should see the effects of concurrency or failures. Our concern is with the *internal consistency* property of transactions' views.

Recent proposals have extended the transaction model to include *nested transactions*, which allow sub-pieces of a transaction to run concurrently and fail independently [Reed78, Moss81]. In such a system the independent failure of (sub)transactions can generate *orphan* processes -- active processes which are running on behalf of a failed transaction. (We will refine and extend this definition below.) Orphans complicate the implementation of atomicity; insuring that orphans' views of the system state are "consistent" with atomicity requires a more sophisticated algorithm than one which ignores orphans' views.

This thesis develops a formal model of a distributed nested transaction system, and it shows that the model satisfies a correctness condition representing "consistency of views." Our transaction system model includes a dynamic *orphan detection* scheme, which detects and exterminates orphans before they see inconsistencies. This model is based on the design for the Transaction Manager of the Argus language under development by the M.I.T. Distributed Systems Group [Liskov82]. Although the models in this thesis simplify both the assumptions made by Argus about the distributed environment, and the specifics of the Argus orphan detection algorithm, the results contribute to confidence in the correctness of this algorithm.

1.1 Nested Transactions

1.1.1 Transactions and Atomicity

An *atomic transaction* is a computation that appears to occur instantaneously and indivisibly from the point of view of any observer of its effects (except for an observer "inside" the transaction). ("Observer" here might refer to another transaction, or to a user of transactions.) If all operations on a system take place through atomic transactions, then each transaction will have the illusion that it is run in isolation: the effects of concurrency are not visible to any transaction. This synchronization property is often referred to as *serializability*: for any observer (including the transactions themselves), the system state seems to be the result of a serial execution of transactions. An execution of transactions can be serializable without being serial (as a trivial example, if no two transactions access the same data objects, then any execution is serializable).

Another property of atomic transactions is *failure atomicity*: each transaction appears to have run completely or not at all. An atomic transaction cannot "partially complete." A transaction which runs to completion is said to "commit;" a transaction which fails (and has no effect) is said to "abort." Failure atomicity simplifies specification of the possible effects of a transaction, since only "good" executions must be considered.

Atomic transactions simplify reasoning about a system because the effects of concurrency and failures can be ignored. Atomicity implies that if an integrity constraint (an *invariant*) on the system state is preserved by all transactions when run in isolation and to completion, then this invariant is preserved by any (possibly concurrent) execution of these transactions. Local, sequential reasoning can be directly applied to a distributed, concurrent environment.

1.1.2 Nested Transactions

Nested transactions extend the usual single-level structure of transactions to a hierarchical structure. A nested transaction can contain other nested (sub)transactions, each of which is atomic with respect to the others. Nesting can be arbitrarily deep. Usual terminology for hierarchical relationships applies to nested transactions. (Thus we refer to the "parent transaction" of a given transaction, or to its "children," etc. "Ancestor" and "descendant" are considered reflexive; "proper ancestor" and "proper descendant" are the corresponding irreflexive relations.)

The child transactions of any transaction can run concurrently; their concurrent execution must be serializable. Children can also commit or abort independently; a child commits to its parent, and its effects will be undone if the parent subsequently aborts. It follows that *permanent* changes to the system state occur only when top-level transactions commit.[1] (For details of the semantics of nested transactions, see [Moss81].)

Nested transactions provide at least three advantages over single-level transactions: The ability to create concurrent children at any level increases the overall parallelism in a system, which might result in efficiency gains. Secondly, the independent abort of a child confines the effects of failure to the work done by that child; the parent can take an appropriate action without aborting itself. This failure isolation improves program robustness and simplifies error recovery. Finally, a program (or a transaction at any level) can use (sub)transactions without regard to their internal concurrency. Concurrency need not be completely specified at the top level, permitting a decentralized design strategy.

1.1.3 Distributed Environment

Two differences between distributed and centralized systems make nested transactions particularly appropriate for distributed environments. First, because distributed systems provide *real* concurrency, a systematic method for managing parallelism becomes both necessary and desirable (for efficiency). Second, the failure modes of distributed systems are much more complex than failure modes of centralized systems because parts of the system can fail independently. For example, one node in a network can go down without affecting other nodes, or the network can fail without directly affecting any node. The nested transaction model allows applications to isolate these failures naturally. (Failure isolation also contributes to node *autonomy*: an application running at one node maintains control over the state at that node even if it spawns subtransactions at other nodes.)

[1] For modifications to remain permanent when nodes crash, each node must provide *stable storage*, and top-level commit must insure that all changes are written to stable storage.

1.2 The Argus System

Although we have attempted to make the models in this thesis relatively general, the Argus system has been used as a starting point. We summarize here the characteristics of Argus which are relevant to this work. Argus is a *programming language* intended to support distributed applications; this language requires an extensive runtime system (for example, to handle transaction management). For details on the language, see [Liskov82].

The distributed environment of Argus consists of a set of *nodes* fully connected in some fashion by a *network*. Nodes can *crash* at any time, and recover after an arbitrary down period. Storage at a node is divided into *volatile* and *stable* storage; the contents of volatile storage are lost when the node crashes, while the contents of stable storage survive crashes.

Nodes communicate by sending *messages* on the network. Delivery of messages is not guaranteed: messages can be lost, duplicated, delayed arbitrarily, and reordered (i.e., delivered in an order other than the order in which they were sent). The network can be partitioned for any period of time. If one node attempts to send a message to another node, it might be unable to distinguish between a lost message, a partitioned network, a crashed respondent, or a respondent that is slow to answer.

Data in the system is partitioned into *objects*; objects are *atomic* or *non-atomic*. We assume that all objects are atomic. (Unconstrained use of non-atomic objects is discouraged in Argus; non-atomic objects are provided as loopholes to allow users to implement atomic types which are more efficient than the "basic" atomic types provided by the system.) While a precise definition of atomic objects is beyond the scope of this thesis (see [Weihl82] for a discussion), we assume that all atomic objects are implemented using two-phase locks with a stack of versions as described in [Moss81]. When an action holds a lock on an atomic object, other unrelated actions are excluded from accessing the object.[2] (Chapter 8 defines a structure which models the lock and version stack of an atomic object.)

Computation is carried out through *actions*, which are atomic transactions. A (sub)action runs completely at one node, though it can spawn child actions at other nodes. Remote subactions are created by a *remote procedure call*, which sends a message from the originating node to the remote node. This message can contain parameters computed at the parent node. If the message is received correctly, the

[2] Moss distinguishes between read and write operations; we will ignore this distinction for simplicity.

subaction runs and can return a message to the parent. The child can commit to its parent, in which case results can be passed back to the parent with a commit message, or it might abort. The parent can abort the child at any time, but this abort is local to the parent's node; the child might still be running at its own node. The parent cannot "commit" the child: the child is committed at the parent's node only if a commit message is received from the child. We say an action *commits to* one of its ancestors if all actions "between" that action and its ancestor commit. We say an action *commits through the top level* if all ancestors of that action commit.

Effects of actions are written to stable storage when their top-level ancestor action commits. A two-phase commit protocol insures that the top-level action commits everywhere or not at all (again, consult [Liskov82] for details). If a node crashes after an action runs there, and that action has committed to its ancestor top-level action, then the crash will be detected during two-phase commit. Thus the top-level action will be aborted. It follows that a crash which undoes the effects of an action (i.e. a crash which precedes the recording of that action's effects on stable storage) guarantees that some ancestor of that action will abort. (This ancestor might not be the top-level ancestor: a lower ancestor might abort, and then the crashed node would not necessarily be checked at two-phase commit.)

1.3 Orphans

An *orphan* is *an active action that is guaranteed not to commit through the top level.* In Argus, orphans can be created in two ways: a proper ancestor can explicitly abort, or a crash can occur.

1.3.1 Creation of Orphans

Argus allows parent actions to unilaterally abort their children, because user requirements might make it unacceptable to wait for confirmation of the abort from the child's node. Complete confirmation would require that each aborted child recursively abort its active children; thus the parent would have to wait until all descendants of the child were halted. Since one of the main reasons for aborting the child might be that the child is not responding (perhaps because of a network partition, or because the child's node crashed), waiting for descendants to be halted could delay the parent indefinitely. Some applications cannot tolerate the possibility of indefinite delay.

Since a parent action can abort a child at the parent's node only, aborted children (and their descendants) might still be active, and might thus be orphans. These orphans are a necessary consequence of a user requirement for bounded delay; they are not the result of a "lazy extermination" strategy.

Orphans result from a node crash when an active action at that node has active descendants at other nodes. This situation is similar to the case of explicit aborts since the active ancestor is effectively "aborted" by the crash. A more complex type of orphan generation occurs when a crash releases a lock held by an action which has committed up to some ancestor, but not through the top level. The lowest active ancestor, and all its active descendants, become orphans since they are guaranteed not to commit through the top level. Since this lowest active ancestor might abort -- or be aborted by its parent -- the crash need not affect higher ancestors. If the lowest active ancestor commits to its parent, the parent and all active descendants of the parent become orphans. If the "infected" ancestor commits to its top-level ancestor, then the crash will be detected during two-phase commit, and the top-level ancestor will abort. This type of orphan could be prevented by keeping locks and versions in stable storage.

1.3.2 Problems Created by Orphans

Orphans are unpleasant, though necessary, side-effects of aborts and crashes. Since their effects are destined to be undone, exterminating orphans cannot do harm. The main concern of this thesis is with the possible adverse consequences of not exterminating orphans "soon enough."

1.3.2.1 Resource Wastage

Orphans consume resources and compete with non-orphans for these resources. Orphans can deadlock with non-orphans, causing non-orphans to be aborted unnecessarily (depending on the deadlock strategy). Resource allocation problems are unlikely to be severe unless orphans are created very frequently. While efficiency issues might be crucial for a working system, this thesis only addresses the semantic problems associated with orphans.

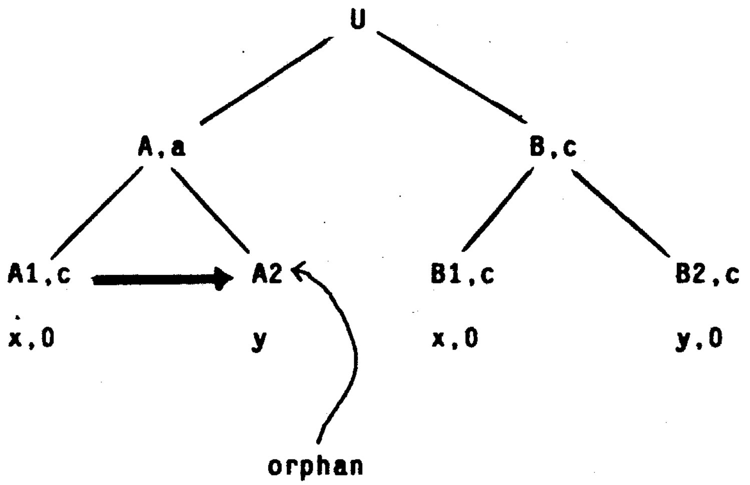
1.3.2.2 Internal Consistency

The transaction management algorithm described in [Moss81] does *not* guarantee atomicity from the point of view of orphans. Orphans can observe system states which are not consistent with serializability (i.e., they can observe the effects of concurrency). Moss's algorithm does not preserve *internal consistency.* The orphan detection algorithm described in the next section is designed to guarantee internal consistency.

We present two examples of such inconsistencies:

1. (See Fig. 1.1. Note that conventions for figures appear in Appendix I.) Initially integers x and y (at different nodes) have values 0. There is an integrity constraint on the system state that x = y. Action A1 runs, reads x = 0, (does not modify x), and commits to A. A then holds a lock on x. (See [Moss81] for a detailed description of the locking protocol.) A then spawns action A2 (passing A2 the information that x = 0), and then A aborts (after the message is sent to create A2), making A2 an orphan. The abort of A releases A's lock on x, allowing B to run to completion and increment both x and y through concurrent children B1 and B2. B commits, releasing its locks on x and y. If A2 (now an orphan) is allowed to read y, it will view y = 1, which allows A2 to infer that x ≠ y (an "inconsistent" view, since x = y will always hold for any serial schedule).

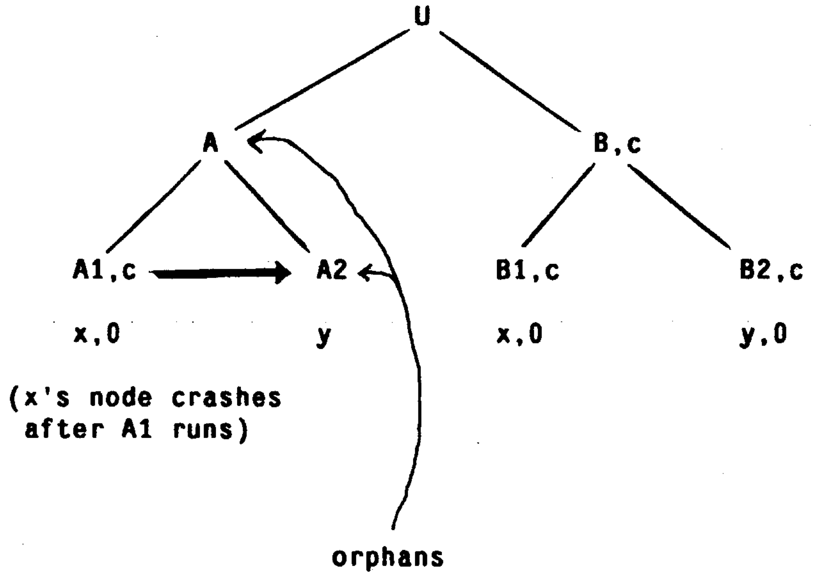
Fig. 1.1. Orphan from Explicit Abort



2. (See Fig. 1.2.) As in the above scenario, integers x and y (at different nodes) have initial values 0, and there is an integrity constraint on the system state that x = y. The same events occur as above: A1 reads x and commits, and A creates A2. Instead of an abort at A, however, x's node crashes. This crash releases A's lock on x, and it makes A (and thus A2) an orphan. As above, B then runs to completion and increments both x and y. B commits, releasing its locks on x and y. If A2 (now an orphan) is allowed to read y, it will view y = 1, which allows A2 to infer that x ≠ y.

It is not clear whether internal consistency is an important concern for a transaction system. One might argue that orphans' views are not important, since orphans will be aborted (eventually) anyway. Since all actions expect a serializable system history, however, programs might function "correctly" only when their views are consistent. Their behavior when views are not consistent might be unpredictable or even catastrophic. (For example, an program guaranteed to terminate under normal conditions might be non-terminating when faced with an inconsistent view.) Orphans could also transmit their inconsistent views to outside parties, via channels which are not under the control of the transaction system. For example, when a user interactively debugs a process that is an orphan, he sees the orphan's (possibly inconsistent) view. This inconsistency might mislead the user, since he might have no direct way of determining that his process is an orphan. A system which permits terminal output by any action

Fig. 1.2. Orphan from Crash



suffers the same problem. (Since terminal output is irreversible, the effects of any aborted action cannot be undone. The orphan's output represents a worse problem, however, since this output might reflect an inconsistent state.)

1.3.3 Orphan Detection Scheme

The basic orphan detection strategy in Argus piggybacks abort and crash information on all channels of information flow between actions. This additional information is used to infer that processes are orphans; these processes are then exterminated.

Our execution model ignores crashes; we deal only with orphans arising from explicit aborts. (We believe that the correctness condition for internal consistency that we develop in Chapter 3 should also apply to a model which includes crashes, although we have not investigated crashes in detail.) We present here a brief description of an orphan detection scheme similar to the portion of the Argus algorithm which handles explicit aborts. The transaction system model we develop is based on this scheme. Our simplified algorithm ignores many of the optimizations envisioned for the actual Argus algorithm.

User programs at nodes communicate via remote procedure calls and returns. In addition to these messages, transaction system messages are sent between nodes to update the status of actions as they commit and abort. Commit and abort messages update the locks and versions of atomic objects. There are many possible strategies for communicating commit and abort information. For example, when an action commits or aborts, a commit message could be sent immediately to all nodes where descendants of that action have run. Alternatively a *querying* strategy could be used where queries are sent about the status of an action only when another action wants a lock held by that action. (The commit and abort messages would then be possible responses to a query.) Our model will not focus on these strategies; we focus on the orphan information which is attached to messages *whenever* these messages are sent. We regard the return message from a remote procedure call as a commit or abort message, depending on whether the child committed or aborted. The return message might include return values, but since our concern is only with orphan information we need not distinguish between return messages and transaction system messages.

Our model has three types of messages: create, commit, and abort messages. A create message models a remote procedure call. Although in Argus a "create" message will only be sent directly from a parent node to a child node, for simplicity we assume that a create message can be sent indirectly through any other node. Communication in our model is very unrestricted; essentially any node can send a message to any other node at any time. The messages that a node can send are limited by what is known at that node (e.g., a node can only send a "commit A" message if it knows that A is committed), and by rules for piggybacking orphan information on messages.

The orphan information at each node is a list, DONE, of known aborts. Any action which is a descendant of an action in DONE is an orphan and is exterminated. Our rules for piggybacking orphan information are quite simple: a create or commit message must include the entire DONE list from its sending node; this list is added to the DONE list at the receiving node when the message is received, and known orphans are exterminated. An abort message need not include any information from DONE.

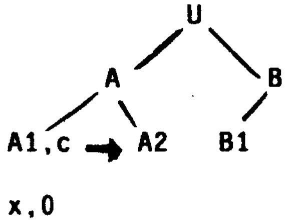
The information flow in this algorithm for the example given in Fig. 1.1 is shown in Fig. 1.3. When A aborts, the abort message releasing A1's lock on x adds A to x's node's DONE list. This DONE list is transmitted to B's node when B1 commits. After B2 runs and commits to B, and B commits, y's node will eventually learn of B's commit. The message that B has committed will contain the DONE from B's node (which now includes A). Thus y's node will know about A's abort. The commit message of B releases B's lock on y, but A2 is now a known orphan at y: A2 is exterminated before it can acquire the lock on y and see an inconsistent state.

The flow of crash information is similar to the flow of DONE information. (We describe the mechanism only superficially here; the actual algorithm is quite complex.) The basic scheme requires each node to maintain a stable *crash count*, which is incremented during recovery from any crash. The orphan information relating to crashes consists of currently known crash counts for nodes plus the crash counts seen by actions when they ran at these nodes. An orphan is detected when it is discovered that a crash count "depended on" by an action (essentially a crash count for a node at which a committed relative has run) is lower than the currently known crash count for the same node. The discrepancy in crash counts implies that a node crash must have occurred since a committed relative ran at that node.

Fig. 1.3. Orphan Detection

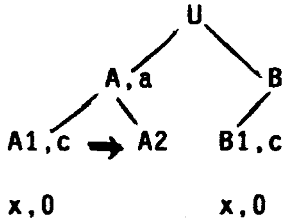
A runs at node M, B at node N
A1,B1 run at node X (object x resides at X)
A2,B2 run at node Y (object y resides at Y)

(1) A1 runs and commits to A; A spawns A2 (A2 has not read y)
B spawns B1; B1 waits because A holds a lock on x



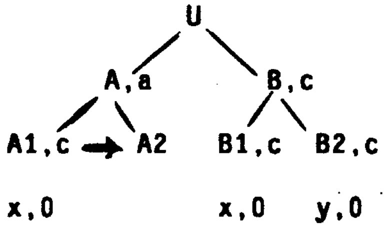
	DONE	LOCKS/VERSIONS
M	Ø	
N	Ø	
X	Ø	x=0, held by A
Y	Ø	y=0

(2) A aborts; abort message sent to X, releasing lock to B1.
B1 increments x and commits; commit message sent to N (with DONE)



	DONE	LOCKS/VERSIONS
M	{A}	
N	{A}	
X	{A}	x=0, held by B
Y	Ø	y=0

(3) B2 runs and increments y and commits. B commits, sending commit message (with DONE) to X and Y. Commit of B arrives at X and Y, releasing B's locks.



	DONE	LOCKS/VERSIONS
M	{A}	
N	{A}	
X	{A}	x=1
Y	{A}	y=1

(4) A2 is aborted because it is a known orphan at Y.

1.4 Related Work

1.4.1 Transaction System

Our transaction system model is based on the design presented in [Moss81]. Moss generalizes two-phase locking for nested transactions, and he develops a recovery scheme based on multiple (backup) versions of objects. His transaction manager functions in the presence of both node crashes and communications failures. He describes distributed algorithms for locking and version restoration, transaction management (including two-phase commit for top-level transactions), and deadlock detection. Although our formal model ignores many of the complexities that Moss considers (in particular, node crashes), it relies heavily on his basic framework.

A different approach to nested transactions is explored by Reed in [Reed78]. This scheme uses timestamps ("pseudo-times") for synchronization rather than using locks. Versions of objects associated with old timestamps can be used for backing up a system to a consistent state. It would be interesting to attempt to extend our models to a timestamp-based scheme such as Reed's. While our lower-level execution models incorporate notions of locks and version stacks, the higher-level models are relatively general, relying only on a nesting relationship among actions and on a notion of "accessing" data.

1.4.2 Orphan Detection Algorithms

As mentioned above, the orphan detection algorithm we consider is based on the orphan algorithm designed for Argus [Liskov82]. Though we are aware of no implementations of orphan detection algorithms, Nelson explores several strategies for eliminating the orphans which result from node crashes [Nelson81]. (Because his design is not based on *atomic* transactions, orphans from broken locks or explicitly aborted ancestors do not arise: his orphans are simply processes running on behalf of ancestors at crashed nodes.) The simplest such strategy is *orphan extermination:* After a node comes back up after a crash, it exterminates all orphans by tracing all outstanding remote calls. As we discussed above, an "immediate extermination" strategy would not be practical for Argus because of user requirements for a bounded delay.

Because communications or node failures can delay extermination during crash recovery indefinitely, Nelson suggests alternate mechanisms which can be used in these (probably rare) cases. Orphan *expiration* requires that a remote call inherit a time limit from its parent; when the time limit is reached the process running the call is automatically killed. Expiration can cause needless failures since processes can be killed even if they are not orphans. The chosen time limit should be significantly longer than "normal" execution times to prevent these anomalies.

Finally, Nelson suggests a scheme which resembles the crash count mechanism in Argus: When complete extermination during crash recovery is delayed, a node will declare a new *epoch* (i.e. increment an "epoch" counter). All messages carry the current known epoch from the sending node. If a node receives a message with an epoch greater than its known epoch, it must either exterminate all currently executing remote calls (assuming that they are orphans), or query the ancestors of remote calls to guarantee that they are not orphans. The system reaches *equilibrium* when all nodes have the same epoch. This approach is most similar to the Argus algorithm because potential orphans are detected *dynamically* based on information piggybacked onto normal information paths.

1.4.3 Formal Models of Atomic Actions

This thesis is a direct extension of the work described in [Lynch82]. Lynch gives the basic definitions for action trees and serializability that we use here. She presents an execution model (at several levels of abstraction) based on Moss's transaction management algorithm, and she shows that these executions satisfy external consistency. Our work extends the correctness condition for executions to include internal consistency, and it modifies the execution models to incorporate orphan detection.

Traditional concurrency control theory generally deals only with single-level transactions. The usual approach is to define a dependency relation among transactions based on reads and updates, and to show that acyclicity of this relation implies serializability (see [Papa79], for example). The basic theory of two-phase locking and serializability for single-level transactions is developed in [EGLT76]; this work forms a basis for Moss's system and hence for our models.

A formal model for nested atomic actions is developed in [Best81]. This model is based on a dependency graph for events, where the notion of "dependency" is left uninterpreted. Atomicity is defined in terms of "collapsing" an event graph to replace a set of events (the events from an "atomic" action) with a single (higher-level) event. Sets of events are configured in a tree structure, representing the nesting relationship of actions. Acyclicity of inter-action dependencies is shown to be sufficient for atomicity. (Lynch uses a *data* dependency relation to derive a similar acyclicity condition for serializability.) The authors also define a condition which they claim is a generalization of two-phase locking, and they show that this condition implies atomicity.

The main difficulty with this dependency graph model is that the graphs cannot be easily related to executions of a transaction system. The action trees developed by Lynch are simply summaries of execution histories; "dependencies" are absent at this level of abstraction. (Although Lynch defines lower level "augmented" action trees which include an ordering on accesses to data, the "dependencies" expressed by this ordering reflect actual modifications to data in an execution sequence.) The advantage of this approach is that Lynch is able to define execution models formalizing a transaction management algorithm, and to prove that her high-level serializability condition is satisfied by these models. This connection between execution models and correctness conditions (for "atomicity") is not explored in [Best81]. We have followed Lynch's approach: we define a condition modeling internal consistency at a high level (the level of action trees), and we develop (at several levels of abstraction) a model of an orphan detection strategy which guarantees this property.

1.5 Outline of the Thesis

Before attempting to show that our orphan detection strategy is correct, we must develop a considerable amount of formal machinery. Chapter 2 presents the basic action tree model as described in [Lynch82]. (Some parts of this chapter are taken directly from [Lynch82]; though these definitions and theorems are not original work of this thesis, we include them here for completeness of presentation.) Serializability is defined for action trees, and a theorem is given relating serializability to acyclicity of data dependencies.

Chapter 3 defines "view-serializability," which models internal consistency. We present a detailed argument explaining why this formal condition corresponds to our intuitive notion of "consistent views." The condition is defined in terms of the action trees and serializability definitions of Chapter 2.

Chapter 4 develops a general execution model for asynchronous systems, the "event-state algebra." We explore a strategy for hierarchical correctness proofs: A correctness condition for executions of a system is defined using a high-level model of its behavior (an algebra); lower-level models are then defined which are progressively closer to the "real" system, and mappings are described between adjacent levels. We also describe *distributed* event-state algebras, which model distributed systems.

Chapters 5 - 10 define successive levels of event-state algebras modeling a transaction system with orphan detection. The correctness condition (view-serializability) appears at Level 0 (the highest level of abstraction). Level 7 (the lowest level of abstraction) is a distributed event-state algebra. At each new level we also construct a mapping to the previous (higher) level.

Chapter 11 summarizes our results, and suggests possible directions for extensions to this work.

2. Action Trees and Serializability

This chapter gives basic definitions and lemmas for action trees and serializability. We define a structure called an "action tree," which is an abstraction of an execution sequence of a nested transaction system. Serializability (and related properties) are expressed as properties of action trees. This approach presents minimal constraints on the implementation of a transaction system since we make few assumptions about the details of concurrency control and recovery algorithms.

2.1 Notation

If S is a set, and o is some order which totally orders the elements of S, then $\langle\langle S;\ o\rangle\rangle$ denotes the sequence consisting of the elements of S in the order given by o.

If S is a set, then $\mathcal{P}(S)$ denotes the powerset of S (the set of all subsets of S).

If S is a set, and $f\colon S \rightarrow \mathcal{P}(S)$, then we associate f with the obvious relation on S $(\{(s,t)\colon t \in f(s)\})$, and we use standard notation for relations. Thus we refer to the *closure* of a set under a function, we describe a function as *acyclic*, etc. f^{+} denotes the transitive closure of f, and f^{*} denotes the reflexive-transitive closure of f.

2.2 Action Summaries and Action Trees

2.2.1 Actions and Objects

Let obj be a universal set of data objects. For each $x \in \text{obj}$, let values(x) denote the set of values x can assume, including a distinguished initial value, init(x). A value assignment is a total mapping $f\colon \text{obj} \rightarrow \text{values(obj)}$, such that $\forall x \in \text{obj},\ f(x) \in \text{values}(x)$.

Let act be a universal set of actions (i.e., transactions). Let U be a distinguished action. We assume that the actions are configured *a priori* into a tree, representing their nesting relationship, with U as the root. For every $A \in \text{act} - \{U\}$, let parent(A) denote the unique parent action for A. Then

$$\text{siblings} = \{(A,B) \in \text{act}^2\colon \text{parent}(A) = \text{parent}(B)\}$$

If $A \in \text{act}$, then children(A) $= \{B \in \text{act}\colon \text{parent}(B) = A\}$. Let top $=$ children(U).

$\text{anc}(A)$ = the set of ancestors of A, $\text{desc}(A)$ = the set of descendants of A

$\text{prop-anc}(A) = \text{anc}(A) - \{A\}$, $\text{prop-desc}(A) = \text{desc}(A) - \{A\}$

For $A \in \text{act} - \{U\}$, define creator(A) as follows:

$A \in \text{top} \Rightarrow \text{creator}(A) = A$

$A \notin \text{top} \Rightarrow \text{creator}(A) = \text{parent}(A)$

If $A,B \in \text{act}$, then let lca(A,B) denote the least common ancestor of A and B. Let

$\text{related} = \{(A,B) \in \text{act}^2 \colon A \in \text{anc}(B) \ \vee \ B \in \text{anc}(A)\}$

$\text{unrelated} = \text{act}^2 - \text{related}$

(Note that $(A,B) \in \text{unrelated} \Rightarrow \text{lca}(A,B) \notin \{A,B\}$.)

If S is a set of actions such that $\forall A,B \in S,\ (A,B) \in \text{related}$, then we say S is an ancestor chain.

If $B \notin \text{anc}(A)$, then let $A \downarrow B$ denote the single element of $\text{anc}(B) \cap \text{children}(\text{lca}(A,B))$. (Note that if $A \in \text{prop-anc}(B)$, then $\text{lca}(A,B) = A$, and $A \downarrow B \in \text{children}(A)$.)

It might be convenient for the reader to think of this *a priori* configuration of all possible actions into a tree as a preassigned "naming scheme" for actions. That is, the "name" of an action is assumed to carry within it information which locates that action in this universal tree of actions. In any particular execution, only some of these possible actions will be "activated." The (virtual) action U, the parent of all top-level actions, has been added for the sake of uniformity.

Let $\text{seq} \subseteq \text{siblings}$ be any fixed partial order, representing sequential dependency. If $(A,B) \in \text{seq}$, then A is constrained to run before B. For the sake of notational simplicity, we are assuming this relation is also fixed *a priori*; we assume that the "name" of any action carries within it information about which siblings the action can assume have completed. The use of an arbitrary partial order is a generalization of both the total order usually specified for the steps which occur within a single-level transaction, and the unconstrained order usually specified among the transactions themselves.

We also assume *a priori* determination of which actions actually access data, which objects they access, and the functions they perform on those objects: Let accesses denote the leaves of the tree described above. (We assume $U \notin \text{accesses}$, so that the set of actions is nontrivial.) Let $\text{object}\colon \text{accesses} \to \text{obj}$ be a fixed function representing which object is read by a particular access. If $\text{object}(A) = x$, we say that A is an access to x, and we write $A \in$ accesses(x). For $A \in$ accesses, let update(A): values(object(A)) $\rightarrow$ values(object(A)) be a fixed function. Let sameobject denote $\{(A,B) \in \text{accesses}^2$: object(A) = object(B)$\}$.

We define the relation of one set of actions *covering* another. This concept will be useful for sets of aborted actions used to detect potentially "harmful" orphans. The covering relation will express the fact that a set has enough information to detect a harmful orphan. Let $R,S \subseteq$ act be any sets of actions; we say S covers R, and we write $R \leq S$ if and only if for each element A in R, there is an ancestor of A in S. The following lemma gives elementary properties of the covering relation:

Lemma 2.2.1.1: Let $R,S,Q,T \subseteq$ act, $A \in$ act, then

a. $R \subseteq S \Rightarrow R \leq S$

b. $\leq$ is transitive: $R \leq S \wedge S \leq T \Rightarrow R \leq T$

c. $(R \leq S \wedge Q \leq T) \Rightarrow R \cup Q \leq S \cup T$

d. $R \leq S \wedge \text{anc}(A) \cap S = \emptyset \Rightarrow \text{anc}(A) \cap R = \emptyset$

Proof: Straightforward from the definition. ∎

2.2.2 Action Summaries

We describe an abstraction of execution sequences, using a structure called an "action summary." An action summary records the status of a particular set of actions (actions can be active, committed or aborted). It also records the data values read by committed accesses. A slightly simpler structure, an "unlabeled action summary" (or UAS) records the same information except for the data values. An "action tree" is any action summary which is a tree:

An action summary, S, has components vertices_S, active_S, committed_S, aborted_S, and label_S, where

- vertices_S is a finite subset of act

- active_S, committed_S, and aborted_S comprise a partition of vertices_S. (These classifications indicate the current status of each known action. When an action is first created, it is classified as active.

At some later time, its classification can be changed to either committed or aborted. By "committed," we mean that the action is committed relative to its parent, but not necessarily committed permanently. Permanent commit of an action would be represented by classification of all ancestors of the action, except for U, as committed.)

- label_S: $\text{datasteps}_S \rightarrow \text{values(obj)}$, (where $\text{datasteps}_S = \text{committed}_S \cap \text{accesses}$), with $\text{label}_S(A) \in \text{values(object}(A))$. (The label of an access to an object is intended to represent the value read by that access. Since the access has an associated function, the value which the access writes into the object is deducible from the value read, and therefore need not be explicitly represented.) The read and update of an access are assumed to occur "instantaneously" when the access commits. (If an access aborts, it has no label because it never sees the object.)

Let $\text{done}_S = \text{committed}_S \cup \text{aborted}_S$. Let $\text{status}_S(A) =$ 'active' (respectively, 'committed', 'aborted') provided $A \in \text{active}_S$ (respectively, committed_S, aborted_S). Let $\text{accesses}_S = \text{vertices}_S \cap \text{accesses}$, $\text{accesses}_S(x) = \text{vertices}_S \cap \text{accesses}(x)$, and $\text{datasteps}_S(x) = \text{datasteps}_S \cap \text{accesses}(x)$. Let $\text{seq}_S = \text{seq} \cap (\text{vertices}_S)^2$. Let $\text{anc-seq}_S = \{(A,B) \in \text{vertices}_S{}^2 \text{: } \exists B' \in \text{anc}(B) \cap \text{vertices}_S \text{: } (A,B') \in \text{seq}\}$. Let $\text{children}_S(A) = \text{children}(A) \cap \text{vertices}_S$.

An unlabeled action summary has all components described above except label_S. An action tree, T, is an action summary where vertices_T is a tree rooted at U: If $A \in \text{vertices}_T - \{U\}$, then $\text{parent}(A) \in \text{vertices}_T$.

If T is an action summary, then unlabel(T) is the UAS obtained by omitting label_T. Definitions and lemmas for UAS's carry over to action summaries in the obvious way (by applying them to unlabel(T)).

2.2.3 Visible and Dead Actions

We describe actions whose existence is intended to be known to other actions (i.e. which are not masked from those other actions by intervening aborts or active actions). We describe these properties for UAS's; corresponding definitions and lemmas hold for (labeled) action summaries and action trees.

Let T be a UAS. For $A \in \text{act}$, let $\text{visible}_T(A) = \{B \in \text{vertices}_T \text{: } \text{anc}(B) \cap \text{prop-desc}(\text{lca}(A,B)) \subseteq \text{committed}_T\}$. That is, $\text{visible}_T(A)$ is just the set of actions whose existence is (potentially) known to A in T, because they and all their ancestors, up to and not including some ancestor of A, have committed. For $A \in act$, $x \in obj$, let $\underline{visible}_T(A,x) = visible_T(A) \cap datasteps_T(x)$. Let $\underline{invisible}_T(A) = vertices_T - visible_T(A)$. The following lemma, which describes elementary properties of "visibility," is proved in [Lynch82]:

Lemma 2.2.3.1: Let T be a UAS, $A,B,C \in act$

a. $A \in desc(B) \wedge B \in vertices_T \Rightarrow B \in visible_T(A)$

b. $A \in visible_T(B) \Leftrightarrow A \in visible_T(lca(A,B))$

c. $A \in visible_T(B) \wedge B \in visible_T(C) \Rightarrow A \in visible_T(C)$

d. $A \in desc(B) \wedge C \in visible_T(B) \Rightarrow C \in visible_T(A)$

e. $A \in desc(B) \wedge B \in vertices_T \wedge A \in visible_T(C) \Rightarrow B \in visible_T(C)$

Actions which are not visible to another action might be masked by an intervening abort, or by active actions only. If B is masked from A by an intervening abort, we say $\underline{B \text{ is dead to } A \text{ in } T}$: if T is a UAS, and $A \in act$, we define $\underline{dead}_T(A) = \{B \in vertices_T\text{: } anc(B) \cap prop\text{-}desc(lca(A,B)) \cap aborted_T \neq \emptyset\}$. Note that $visible_T(A) \cap dead_T(A) = \emptyset$. If $A \in act$, $x \in obj$, then $\underline{dead}_T(A,x) = dead_T(A) \cap datasteps_T(x)$. If B is not dead to A in T, we say that $\underline{B \text{ is live to } A \text{ in } T}$. If $A \in vertices_T$, then we say $\underline{A \text{ is live in } T}$ iff $anc(A) \cap aborted_T = \emptyset$, and $\underline{A \text{ is dead in } T}$ otherwise. If T is a UAS, $A \in vertices_T$, and A is dead in T, then we define the $\underline{\text{crucial abort of } A \text{ in } T}$, denoted $\underline{crucial}_T(A)$, as the lowest aborted ancestor of A in T: i.e., if $S = anc(A) \cap aborted_T$, then $crucial_T(A) \in S$, and $\forall B \in S$, $crucial_T(A) \in desc(B)$. (If A is not dead in T, then $crucial_T(A)$ is undefined. In this case we will consider that $\{crucial_T(A)\} = \emptyset$, for convenience.)

Let T be a UAS, $A \in vertices_T$, then we define

$\underline{v\text{-}seq}_T(A) = \{B\text{: } (B,A) \in seq \wedge B \neq A\} \cap visible_T(A)$

$\underline{i\text{-}seq}_T(A) = \{B\text{: } (B,A) \in seq \wedge B \neq A\} \cap invisible_T(A)$

$\underline{v\text{-}anc\text{-}seq}_T(A) = \{B\text{: } (B,A) \in anc\text{-}seq_T \wedge B \notin anc(A)\} \cap visible_T(A)$ (see Fig. 2.1.)

$\underline{i\text{-}anc\text{-}seq}_T(A) = \{B\text{: } (B,A) \in anc\text{-}seq_T \wedge B \notin anc(A)\} \cap invisible_T(A)$ (see Fig. 2.1.)

$\underline{\text{v-child}}_T(A) = \text{children}(A) \cap \text{visible}_T(A)$

$\underline{\text{i-child}}_T(A) = \text{children}(A) \cap \text{invisible}_T(A)$

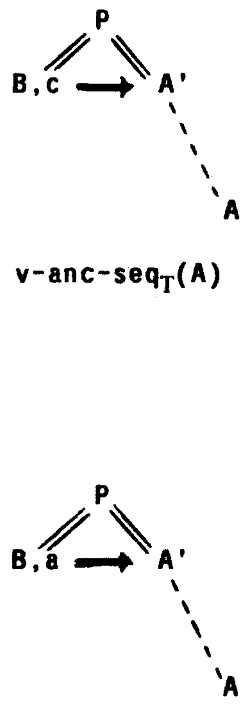
$\underline{\text{v-desc}}_T(A) = \text{desc}(A) \cap \text{visible}_T(A)$

$\underline{\text{i-desc}}_T(A) = \text{desc}(A) \cap \text{invisible}_T(A)$

2.3 Augmented Action Trees

We define a new structure called an augmented action summary (or AAS). We can regard AAS's as action summaries with an additional component: an ordering on the datasteps accessing each object. Formally we define an AAS as a pair $T = \langle S,\mathcal{O}\rangle$, where S is an action summary, and $\mathcal{O}$: obj → $\mathfrak{P}$(sameobject), where for all $x \in$ obj, $\mathcal{O}(x)$ is a total order on $\text{datasteps}_S(x)$. (Thus $\mathcal{O}(x) \subseteq \text{datasteps}_S^2(x)$.) If $T = \langle S,\mathcal{O}\rangle$ is an AAS, then we define $\underline{\text{erase}}(T) = S$, $\underline{\text{order}}(T) = \mathcal{O}$. We extend our notation for

Fig. 2.1. Visible and Invisible Ancestor-Sequence



B ∈ v-anc-seq$_T$(A)

(or B could be active)

B ∈ i-anc-seq$_T$(A)

components and functions of action summaries to components and functions of AAS's in the obvious way, by applying them to erase(T). (For example if T = $\langle S,O\rangle$, then we will use "vertices$_T$" to refer to vertices$_S$.) Definitions for action summaries and UAS's carry over to AAS's in the obvious way (by applying them to erase(T) or to unlabel(erase(T))). If T = $\langle S,O\rangle$, then we define data$_T$ = $\bigcup_{x \in obj} O(x)$. An augmented action tree (AAT) is an AAS where erase(T) is an action tree.

Let T be an AAS, A $\in$ vertices$_T$, then we define

v-data$_T$(A) = $\{B: (B,A) \in data_T \wedge B \neq A\} \cap visible_T(A)$

i-data$_T$(A) = $\{B: (B,A) \in data_T \wedge B \neq A\} \cap invisible_T(A)$

v-data-anc$_T$(A) = $\bigcup_{B \in v\text{-}data_T(A)} \{A \downarrow B\}$

i-data-anc$_T$(A) = $\bigcup_{B \in i\text{-}data_T(A)} \{crucial_T(B)\}$

v-precedes$_T$(A) = v-anc-seq$_T$(A) $\cup$ v-child$_T$(A) $\cup$ v-data-anc$_T$(A)

i-precedes$_T$(A) = i-anc-seq$_T$(A) $\cup$ i-child$_T$(A) $\cup$ i-data-anc$_T$(A)

The "visible precedence" relation, v-precedes$_T$, will be used in Chapter 3 to define a "view tree" which represents an action's view of an execution history. We state here some elementary properties of this relation.

Lemma 2.3.1: Let T be an AAS, A $\in$ vertices$_T$. Then
B $\in$ v-precedes$_T$(A) $\Rightarrow$ parent(B) = lca(A,B).

Proof:

1. B $\in$ v-anc-seq$_T$(A) $\Rightarrow$ (B,A') $\in$ seq for some A' $\in$ anc(A), and B $\neq$ A'. Thus parent(B) = parent(A') = lca(A,B).
2. B $\in$ v-child$_T$(A) $\Rightarrow$ parent(B) = A = lca(A,B).
3. B $\in$ v-data-anc$_T$(A) $\Rightarrow$ B = A$\downarrow$b for some b $\in$ accesses. Thus B $\in$ children(lca(A,B)), $\Rightarrow$ parent(B) = lca(A,B). ∎

Lemma 2.3.2: Let T be an AAS, A $\in$ vertices$_T$. Then B $\in$ v-precedes$_T^+$(A) $\Rightarrow$

$B \in \text{visible}_T(A)$, and $B \in \text{committed}_T$.

Proof: $B \in \text{visible}_T(A)$ is obvious from transitivity of visible_T (Lemma 2.2.3.1c). To see that $B \in \text{committed}_T$, note that if $B \in \text{visible}_T(C)$ for some C, then $B \in \text{committed}_T$ or $B \in \text{anc}(C)$.

But $B \in \text{v-precedes}^+_T(A) \Rightarrow B \in \text{v-precedes}_T(C)$ for some C, $\Rightarrow B \in \text{visible}_T(C)$. But $B \notin \text{anc}(C)$, by Lemma 2.3.1, so $B \in \text{committed}_T$. ▮

If T is an AAS, $A \in \text{vertices}_T$, then we define the view set of A in T as the v-precedes_T-closure of A: $\text{vset}_T(A) = \text{v-precedes}^*_T(A)$. The following lemma gives elementary closure properties of view sets.

Lemma 2.3.3: Let T be an AAS, $A \in \text{vertices}_T$, $B \in \text{vset}_T(A)$. Then

a. $\text{vset}_T(B) \subseteq \text{vset}_T(A)$

b. $\text{v-desc}_T(B) \subseteq \text{vset}_T(A)$

c. $\text{v-data}_T(B) \subseteq \text{vset}_T(A)$

Proof: (a) is obvious from the definition. v-desc_T-closure (b) follows inductively from $\text{v-child}_T \subseteq \text{v-precedes}_T$. We show (c):

Suppose $C \in \text{v-data}_T(B)$. Then $B\downarrow C \in \text{v-data-anc}_T(B)$
$\Rightarrow B\downarrow C \in \text{vset}_T(A)$, since $\text{vset}_T(A)$ is v-precedes_T-closed.
$C \in \text{visible}_T(B) \Rightarrow C \in \text{v-desc}_T(B\downarrow C)$.
But $\text{vset}_T(A)$ is v-desc_T-closed by (b), $\Rightarrow C \in \text{vset}_T(A)$. ▮

The following lemma gives an ancestor-closure property for view sets (the view set itself is not ancestor closed, but the view set of an action together with the proper ancestors of that action forms an ancestor-closed set).

Lemma 2.3.4: Let T be an AAS, $A \in \text{vertices}_T$. If $W = \text{vset}_T(A) \cup \text{prop-anc}(A)$, then W is anc-closed.

Proof: Let $V = \text{vset}_T(A)$. We show inductively that $B \in V \Rightarrow \text{anc}(B) \subseteq W$. Since $B \in$

$\text{prop-anc}(A) \Rightarrow \text{anc}(B) \subseteq \text{prop-anc}(A) \subseteq W$, anc-closure follows.

Basis: $A \in W$, $\text{anc}(A) \in \{A\} \cup \text{prop-anc}(A)$.

Induction: Assume $B \in V$, $\text{anc}(B) \subseteq W$, and take $C \in \text{v-precedes}_T(B)$. By Lemma 2.3.1, $\text{parent}(C) = \text{lca}(B,C) \Rightarrow \text{prop-anc}(C) \subseteq \text{anc}(B) \subseteq W$. But $C \in V \Rightarrow \{C\} \subseteq W \Rightarrow \text{anc}(C) \subseteq W$. ∎

2.4 Serializability

We define serializability for action trees. Let T be an action tree. A partial order $p \subseteq$ siblings is linearizing for T provided p totally orders all siblings in T. A linearizing partial order p induces a total order, $\text{induced}_{T,p}$, on accesses_T, in the obvious way: $(A,B) \in \text{induced}_{T,p} \Leftrightarrow (B\!\downarrow\!A, A\!\downarrow\!B) \in p$. If $A \in \text{accesses}_T(x)$ and p is a linearizing partial order for T, let $\text{preds}_{T,p}(A)$ denote the sequence $\langle\langle\{B \in \text{visible}_T(A,x): (B,A) \in \text{induced}_{T,p} \land B \neq A\}; \text{induced}_{T,p}\rangle\rangle$.

If $x \in$ obj and s is some finite sequence of accesses, then we define result(x,s) as follows: If s is the empty sequence, then result(x,s) = init(x). Otherwise let s = s'A, where $A \in$ accesses. Then result(x,s) = update(A)(result(x,s')) if A is an access to x, = result(x,s') otherwise.

A linearizing partial order p for T is said to be a serializing partial order for T provided p is consistent with seq, and $\text{label}_T(A) = \text{result}(x,\text{preds}_{T,p}(A))$, for all $A \in \text{datasteps}_T(x)$. This definition says that the value seen by each datastep is equivalent to the result of a serial execution in the order given by p, where only committed actions have any affect. T is said to be serializable provided there exists some serializing partial order for T.

2.5 Serializability of Augmented Action Trees

An AAT, T, is serializable iff erase(T) is a serializable action tree. It is convenient to define a stronger condition than serializability for AATs, which we call "data-serializability." An AAT, T, is data-serializable iff there exists p, a serializing partial order for erase(T), with the additional property that $\text{induced}_{T,p}$ is consistent with data_T. Obviously if T is data-serializable, then it is serializable.

Data-serializability has a cycle-free characterization similar to those in usual concurrency control theory. First, we give a definition which says that the label of each access describes the correct object value which the access should see, if the versions of objects are ordered according to the data_T order. Formally, an AAT is version-compatible iff for every object $x \in \text{obj}$, and every $A \in \text{datasteps}_T(x)$, it is the case that $\text{label}_T(A) = \text{result}(x,s)$, where $s = \langle\langle \text{v-data}_T(A); \text{data}_T \rangle\rangle$. The following theorem is proved in [Lynch82]:

Theorem 2.5.1: An AAT, T is data-serializable if and only if both of the following are true:

a. T is version-compatible.

b. There are no cycles of length greater than one in $\text{seq}_T \cup \text{sibling-data}_T$.

2.6 Restrictions of Trees

It is often useful to project an action tree (or an AAT) onto a particular set of vertices. We call the resulting action summary a *restriction* of the original tree.

Defn 2.6.1: Let T be an action tree (or an AAT), $V \subseteq \text{vertices}_T$. We define the restriction of T to V, denoted $T|V$, as follows: (let $S = T|V$)

$\text{vertices}_S = V$
$\forall v \in V,\ \text{status}_S(v) = \text{status}_T(v)$
$\forall A \in \text{datasteps}_S,\ \text{label}_S(A) = \text{label}_T(A)$

If T is an AAT, then $\text{data}_S = V^2 \cap \text{data}_T$

We say S is a restriction of T iff $S = T|\text{vertices}_S$. We say S is a subtree of T iff S is a restriction of T which is also a tree rooted at U (i.e. vertices_S is anc-closed).

Stating the simplest correctness requirements for executions only requires consideration of actions whose effects become "permanent." For an action tree (or AAT), T, we define a restriction of T to all actions which have committed through the top level: $\text{perm}(T) = T|\text{visible}_T(U)$. It is easy to verify that perm(T) is a subtree of T.

The following lemma shows that if an action has no descendants in datasteps_T, then it cannot affect serializability of T:

Lemma 2.6.2: Let T be an action tree, $A \in \text{vertices}_T - \{U\}$. If $\text{desc}(A) \cap \text{datasteps}_T = \emptyset$, then T is serializable if and only if $T|(\text{vertices}_T - \text{desc}(A))$ is serializable.

Proof: Let $T' = T|(\text{vertices}_T - \text{desc}(A))$.
First we show T serializable $\Rightarrow$ T' serializable. Let p be a serializing partial order for T, and let p' be p restricted to $\text{vertices}_{T'}$. Then p' is obviously a linearizing partial order for T'. Let $B \in \text{datasteps}_{T'}$.

$\text{label}_T(B) = \text{label}_{T'}(B) = \text{result}(x, \text{preds}_{T,p}(B))$, since p is serializing for T. But $\text{desc}(A) \cap \text{datasteps}_T = \emptyset, \Rightarrow \text{preds}_{T',p'}(B) = \text{preds}_{T,p}(B)$. Thus p' is a serializing order for T'.

Now assume T' is serializable, and let p' be a serializing partial order for T'. Let p be any linearizing order for T that is consistent with p'. Let $B \in \text{datasteps}_T$. Then $B \in \text{datasteps}_{T'}$.

$\text{label}_T(B) = \text{label}_{T'}(B) = \text{result}(x, \text{preds}_{T',p'}(B))$, since p' is serializing for T'. But $\text{desc}(A) \cap \text{datasteps}_T = \emptyset, \Rightarrow \text{preds}_{T',p'}(B) = \text{preds}_{T,p}(B)$, since p is consistent with p'. Thus p is a serializing order for T. ∎

We will frequently use trees that are restrictions of the global action tree with the exception that the proper ancestors of one action are considered active (instead of whatever status they have in the global action tree). We term this process "backing up" an action tree since we are effectively undoing whatever commits or aborts of the proper ancestors might have occurred. This construction will be useful for defining trees representing the "view" of an action, since the action will believe its proper ancestors to be active (whether or not they have already committed or aborted).

Defn 2.6.3: Let T be an action tree (or an AAT), $A \in \text{vertices}_T$. We define the tree T backed up through A, denoted T//A, as follows: (let S = T//A)

$\text{vertices}_S = \text{vertices}_T$
$B \in \text{prop-anc}(A) \Rightarrow \text{status}_S(B) = \text{'active'}$
$B \in \text{vertices}_T - \text{prop-anc}(A) \Rightarrow \text{status}_S(B) = \text{status}_T(B)$

$\forall A \in \text{datasteps}_S,\ \text{label}_S(A) = \text{label}_T(A)$

If T is an AAT, then $\text{data}_S = \text{data}_T$

Finally, for functions from actions to sets of actions we will occasionally want to exclude some actions from the domain of a function. The set of actions excluded will always be the proper ancestors of a particular action, so we define exclusion with respect to this action:

Defn 2.6.4: Let f: act → $\mathcal{P}$(act). We define the exclusion of f from A, denoted f//A, as the function:

$$\begin{aligned}(f//A)(s) &= f(s), \text{ if } s \notin \text{prop-anc}(A),\\ &= \varnothing, \text{ if } s \in \text{prop-anc}(A)\end{aligned}$$

3. View-Serializability

This chapter presents a correctness condition for action systems, which we call view-serializability. The definitions relating to view-serializability are developed using action trees: no specific execution model for generating these trees is yet assumed. View-serializability is intended to model "internal consistency:" a system which generates only view-serializable action trees will not allow actions to see inconsistent states, even if these actions are orphans.

3.1 External Consistency and Internal Consistency

A fundamental property of atomic actions is that the effects of their concurrent execution should be "equivalent to" an execution where each action is run in isolation, and (if the action commits) to completion. Different notions of "equivalence" give rise to different conditions modeling atomicity. *External* consistency of a transaction system requires that for any execution the view of an observer *outside* the system is identical to the view that would result from some serialization of this execution. There might be interaction between an action and a user which is outside the scope of the "system" (e.g. output to a terminal, which cannot be undone when an action aborts). Since a transaction system can only make guarantees about the states of objects under system control, we will ignore the effects of "extra-system" communication on serializability. (Insuring consistency in such an environment is the responsibility of user programs. At this level, "consistency" is an application-specific concept: for some applications terminal output from actions which are later aborted might be acceptable, for example.) Given this restriction, only actions which commit through the top level can affect the system state as seen by an outside observer.

Internal consistency requires that the effects of concurrency are masked from any action in the system. If a system provides external consistency, then all actions which commit through the top level must see system states consistent with some serial schedule. Other actions might see inconsistent states, however. In particular, the views of orphans are not considered for external consistency, since orphans cannot commit through the top level.

We model external consistency by requiring that perm(T), the subtree of the action tree consisting of all actions which commit through the top level, be serializable. In [Lynch82], a model for a distributed transaction system based on the locking protocol developed in [Moss81] is shown to be externally consistent: Lynch shows that for all action trees, T, generated by the model, perm(T) is serializable.

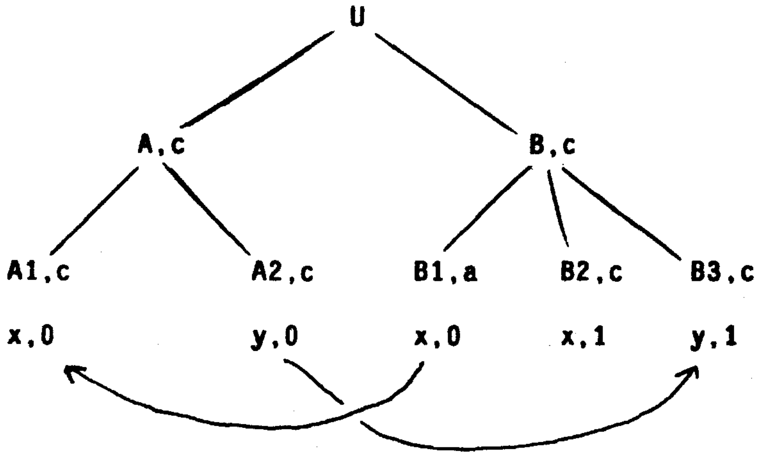
To see that serializability of perm(T) is not sufficient to guarantee internal consistency, consider the example from Fig. 1.1 The consistency constraint x = y is violated for action A2, but perm(T) (which consists of U, B, B1, and B2) is serializable.

Although serializability of perm(T) is not sufficient for internal consistency, serializability of the entire action tree is not necessary for internal consistency. We can easily construct action trees for executions which we believe are internally consistent (since no action can see an inconsistent state), but which are not serializable. Consider the example shown in Fig. 3.1. Again, the integrity constraint on the system state is x = y. Initial values of x and y are 0. Action B1 runs first, views x = 0, and then aborts. Then actions A1,A2,B2, and B3 run (in that order). A1 and B2 increment x, and A2 and B3 increment y. The tree is not serializable, because A must be serialized before B (since B2 views x = 1), yet B1 did not view the effect of A1. The tree is internally consistent, however, because no particular action was able to observe x ≠ y. (B1 viewed x = 0, but it had no information about the value of y. Since B1 aborted, it did not pass its view of x to the rest of B.)

Thus serializability of the entire action tree is too strong a condition for internal consistency. We need a weaker condition which takes into account the views of aborted actions and orphans as well as the views of actions that commit through the top level.

In the following sections we will define the possible "views" of each action in an action tree, and we will state a condition modeling internal consistency which is based on serializability of these views.

Fig. 3.1. Non-serializable, Internally Consistent Action Tree



Using this definition, the only view for action U will be perm(T); thus our formal "view" for action U corresponds to our intuitive notion of the view of an "outside observer." Our condition for external consistency will then be a special case of our condition for internal consistency.

3.2 Information Flow and Information Trees

3.2.1 Information: Object Values and Execution Histories

Internal consistency requires that any action's view of the system state must be consistent with an "illusion" of serial execution. To formalize internal consistency, we must attempt to be precise about what constitutes an action's "view" in a particular system state. A simple approach would try to capture the knowledge that an action has of the current *values of objects.* Thus for the example in Fig. 1.1, we might say that action A2 knows that $x=0$, and if A2 is allowed to read y then it will know that $x=0$ and that $y=1$ (an "inconsistent" view).

A definition which describes the view of an action as a (partial) binding of objects to known values is not sufficient to handle more complex examples, however. Suppose that action A creates concurrent children A1 and A2 to read and update object x. x is a boolean object, assuming only logical values (0 and 1). Both A1 and A2 read x, and perform a logical *not* operation on x. A1 returns the value 0, and A2 returns the value 1. If A cannot determine which child ran first, then it is unsure of the "current" value of x.

This uncertainty about "current" values can affect our notion of "consistency." Suppose, for example, that action C creates child C1 to read object x, and C1 returns the value $x=1$. C then creates concurrent children C2 and C3, passing them the "information" that $x=1$. But C2 and C3 both read and increment x. Depending on which action runs first, the later one will see an "inconsistency" between what its parent told it ($x=1$) and the current state ($x=2$). But if both C2 and C3 realize that the other might have run first, then both can explain this potential "inconsistency."

These examples illustrate that direct information about the "current" value of an object is only available to accesses which directly read that object. All other information is "hearsay," in a sense, because it expresses only what *another* action saw or was told. We thus regard the "information" available to an action as its knowledge of the *execution history* of the system: an action might know with certainly that action B read $y=5$, but it cannot automatically assume that the value of y *is* 5. By treating information as information about execution histories, we can explain the seeming ambiguities and conflicts described in the examples above. For the first example, A's information is that A1 read x=0 and A2 read x=1. In the second example, the information available to C2 and C3 is that "C1 ran and saw x=1." Neither C2 nor C3 can conclude that the current value of x is 1. If C2 were run sequentially before C3 (and C had no other children), then C2 could conclude x=1. This conclusion of C2 depends on a serializability assumption, which is a basic part of consistency, and on C2's knowledge of the structure of other actions (in this case knowledge that no siblings can intervene between C1 and C2). We elaborate on these points in the following sections.

3.2.2 Paths of Information Flow

In designing system algorithms to guarantee consistency, we often take a "worst case" approach regarding information flow among actions. To define an action's view in this sense, we must consider all possible sources of information about the execution history to an action. We say that *information flows from action A to action B* if B learns something about the execution history from A. The actual value(s) passed from A to B will generally be some function of the values of objects seen by A; we lose no generality by assuming that A passes B its complete knowledge of the execution history. Again, this assumption amounts to a worst-case approach for information flow: If action A reads object x, and A passes some information to B, B does not necessarily have specific "information" about the value of x seen by A. The actual values passed from A to B might be constants, for example, giving B no information at all about the execution history. But since B *might* have any information that A might have had, we will assume that it does.

Let A be an action whose view is being defined. We imagine that actions are encapsulated in procedure-like structures, with well-defined inputs and outputs. Thus we assume that information can flow to A only in the following three ways:

1. If A is an access to x (and A commits), then A reads the value of x.

2. parent(A) passes information to A when A is created.

3. Committed children of A pass information to A when they return (i.e. when they commit to A).

Path (3) is limited to *committed* children, reflecting an assumption that aborted children do not pass "information" to their parents. If aborted children are allowed to return values to their parents (as in Argus), then this assumption can be violated. In Argus, return values from aborted children are a recognized "loophole" in the system. We retain our assumption because it models the fundamental semantics of "abort" which are derived from atomicity: An atomic action runs completely or not at all. If an atomic action aborts, all effects should be *as if* it had never run at all, and an action which never runs cannot return values.

A more subtle assumption is that the very fact that a child has aborted cannot give the parent any "information" about the execution history, other than the fact that the child aborted. A child which reads object x might be programmed to commit if it sees $x = 1$, for example, and to abort otherwise. If the child aborts, one might think that the parent could then assume that the child read x and found $x \neq 1$. However, we make a basic assumption that an action can be aborted at any time *by the system*, and that the parent cannot necessarily distinguish between a system-initiated abort and an abort caused by the child itself. For example, the system might abort a child because of a communications failure, even if the child were going to commit. (In a practical system, such as Argus, it might be useful to identify the cause for a system-initiated abort, so the parent will know how to proceed. These explanations for aborts fall into the same "loophole" category as return values from aborted children.) Given the assumptions that aborted children cannot return values, and that aborts are always possible, whatever the system state, aborts serve as impenetrable barriers to information flow.

3.2.3 Circularity of Information Flow

We would like to describe the information available to an action in an action tree by listing all the actions which are (potential) sources of information to that action. Our formulation of the three paths of information flow is not convenient for this purpose, because it contains a confusing circularity: information flows from a parent to its children, and also from a (committed) child back to its parent. By naively following the paths of information flow we would conclude that an action is a source of information to itself, which makes no sense. Of course this circularity is fictitious, because the flow of information from parent to child happens at a different *time* than the flow of information from child to parent.

One approach based directly on the three paths of information flow above would be to define the information available to an action as a function of time. By including time as a parameter of available information, the circularity described above can be removed (i.e. "available information" will no longer be recursively defined). We would like to describe the information available to an action without referring to time, however. Although the information available to an action does change as an execution proceeds, we would like to capture the *maximum* amount of information that an action sees during an execution. Since an action's information can only increase over time, an action attains its maximum information at its *latest active point* in an execution (if it completes, this point is immediately before it commits or aborts).

We achieve a "time-independent" definition of available information below by reformulating the paths of information flow. The alternate formulation contains no circularity.

3.2.4 Information Flow from Siblings of Ancestors

We remove the circularity in the paths of information flow by "short-circuiting" flow through ancestors:

Information can flow between sibling actions via the parent only if one sibling commits before another is created: upon commit, the first sibling passes information to its parent, and the parent passes this information to the next sibling when creating it. (There can also be indirect information flow via objects.) In some systems, this path of information flow might allow an action to see information known by *any* sibling which had committed before the action was created. We assume that flow of information between siblings (via the parent) is restricted to flow from *sequentially preceding* committed siblings. We are making an assumption here that the control structure of actions does not permit direct flow of information between concurrent siblings. (This assumption holds in Argus, because all concurrent siblings must be created "at once" by a **coenter** statement. It is impossible for a concurrent sibling to commit before another is created; thus it is impossible for information to flow directly between them. Concurrent siblings cannot communicate except by modifying shared objects.)

Thus we can list the the sources of information to A's parent which can serve as sources of information to A (when A is created): (1) Sequentially preceding committed siblings of A, (2) Any action which was a source of information to A's parent when the *parent* was created. In "unwinding" this recursion, we can define the sources of information to A when A is created as all actions which are committed and sequentially precede some ancestor of A. We thus obtain an equivalent definition of the sources of information to an action by replacing (2) above with a path of information flow from these sequentially preceding committed siblings of ancestors:

2. Information passed from B to A, where B has committed, and B sequentially precedes some ancestor of A ($B \in$ v-anc-seq_T(A)).

Using this second formulation we can give a *single* definition of the (maximum) information available to an action in any particular execution history (i.e. for any action tree). With the new specification of information source (2), the only paths of information flow are from *committed* actions (and from objects). We assume that committed actions release their complete (maximum) information to other actions when they commit.

3.2.5 Information Trees

Since we are using action trees as an abstraction of execution histories (and hence of system states), we describe an action's view of the history as a particular (backed up) subtree of the (global) action tree. We call this tree the information tree for an action. We can think of the information tree for action A as being defined recursively: it is constructed by merging all the information trees of actions from which information can flow to action A.

Because an action might be aware that some actions have aborted, these aborts should strictly be included in the information tree. (If action A sequentially precedes B, for example, then B will know that A has either committed or aborted.) Although aborts are part of the execution history, we have argued above that they convey no additional information. (In other words, the existence of an abort tells an action *nothing* other than that the abort occurred.) For simplicity, then, we exclude these aborted actions from the information tree.

The vertices of the information tree for an action are simply all vertices reachable by "tracing back" the three paths of information flow listed above. Since the information tree is a subtree of the global action tree, path (1) is accounted for by the labels of datasteps. (In other words, if a datastep is labeled with "u" in the global action tree, it will be labeled with "u" in the information tree. This value read is part of the execution history of the datastep, and should thus be included with the datastep.) Path (2) requires that if B is in the information tree, and $C \in$ v-anc-seq_T(B), then C is in the information tree. Path (3) requires that if B is in the information tree, and C is a committed child of B, then C is in the information tree.

Defn 3.2.5.1: Let T be an action tree, $A \in \text{vertices}_T$. We define the information set of A in T,

$\underline{\text{info-set}_T}(A) = (\text{v-anc-seq}_T \cup \text{v-child}_T)^*(A)$

And we define the information tree of A in T,

$\underline{\text{info-tree}_T}(A) = (T|W)//A$, where $W = \text{info-set}_T(A) \cup \text{prop-anc}(A)$

We include proper ancestors of A in the information tree, but since information has only flowed *through* these ancestors from sequentially preceding committed siblings, we do not include them in the information set. The proper ancestors are considered active since A will regard them as active. (Thus the information tree is "backed up" through A.) It is possible that some of these ancestors might have committed or aborted, but these changes in status should not be visible to A.

The following lemma gives an equivalent definition of the information set which is easier to use because it does not involve closures of functions.

Lemma 3.2.5.2: Let T be an action tree, $A \in \text{vertices}_T$. Then

$\text{info-set}_T(A) = \text{v-desc}_T(\text{v-anc-seq}_T(A) \cup \{A\})$.

Proof: Let $V = \text{info-set}_T(A) = (\text{v-anc-seq}_T \cup \text{v-child}_T)^*(A)$, and let $W = \text{v-desc}_T(\text{v-anc-seq}_T(A) \cup \{A\})$. It is obvious that $W \subseteq V$. We show $V \subseteq W$ by induction on V:

Basis: $A \in V$, but $A \in W$ because $A \in \text{v-desc}_T(\{A\})$.

Induction: Let $B \in V$, and assume $B \in W$. Take $C \in \text{v-child}_T(B) \cup \text{v-anc-seq}_T(B)$. We show $C \in W$.

Since $B \in W$, $B \in \text{v-desc}_T(B')$, for some $B' \in \text{v-anc-seq}_T(A) \cup \{A\}$. If $C \in \text{v-child}_T(B)$, then $C \in \text{v-desc}_T(B')$. If $C \in \text{v-anc-seq}_T(B)$, then either $C \in \text{prop-desc}_T(B')$, or $(C,B') \in \text{siblings}$, or $C \in \text{v-anc-seq}_T(\text{parent}(B'))$.

If $C \in \text{prop-desc}_T(B')$, then $C \in \text{v-desc}_T(B')$.
If $(C,B') \in \text{siblings}$, then $(C,B') \in \text{seq}$, $\Rightarrow$ $C \in \text{v-anc-seq}_T(A)$, by transitivity of seq.

If C ∈ v-anc-seq$_T$(parent(B')), then C ∈ v-anc-seq$_T$(A). ▮

We now use this equivalent definition of the information set to prove three simple lemmas about information sets and information trees:

Lemma 3.2.5.3: Let T be an action tree, A ∈ vertices$_T$, W = info-set$_T$(A) ∪ prop-anc(A). Then W is ancestor-closed. (Thus the information tree is in fact a tree.)

Proof: Let B ∈ W, C ∈ prop-anc(B). We must show C ∈ W. Let V = info-set$_T$(A). If B ∈ prop-anc(A), then C ∈ prop-anc(A) ⇒ C ∈ W.
If B ∈ V then B ∈ v-desc$_T$(v-anc-seq$_T$(A) ∪ {A}), by Lemma 3.2.5.2. If B ∈ v-desc$_T$({A}), then C ∈ v-desc$_T$({A}) ∪ prop-anc(A) ⇒ C ∈ W.
If B ∈ v-desc$_T$(v-anc-seq$_T$(A)), then either C ∈ v-desc$_T$(v-anc-seq$_T$(A)), or C ∈ anc(A), ⇒ C ∈ W. ▮

Lemma 3.2.5.4: Let T be an action tree, A ∈ vertices$_T$. Then prop-anc(A) ∩ info-set$_T$(A) = ∅.

Proof: Follows directly from Lemma 3.2.5.2. ▮

Lemma 3.2.5.5: Let T be an action tree, A ∈ vertices$_T$, and let S = info-tree$_T$(A). Then vertices$_S$ ⊆ visible$_T$(A).

Proof: Follows directly from Lemma 3.2.5.2. ▮

3.3 Behavioral Constraints and View Trees

The information tree represents all information about the execution history which might be available to a particular action as a result of information flow in this execution (except for information about aborts.) For this information to be "consistent," it must not contradict the assumptions an action might have about the system's behavior. One of these assumptions is the *illusion* of serial execution: no action should see the effects of concurrency. Failure atomicity also requires that no action should see the effects of aborted actions. An action might have additional expectations about the system's behavior, however. Often these expectations are captured in *invariants* on the system state which all actions preserve (when run in isolation and to completion). An action might function correctly only if a particular invariant holds. (Its effects when the invariant does not hold might be unexpected or unspecified.)

To develop a notion of "consistency," we imagine that an observer is placed at an action and is given that action's information tree. The observer is also informed of any invariants on the system state that are preserved by all actions in isolation, and he is told that the system executes actions in some serial order. (Of course, the actual order might not be serial, but the observer should be unaware of this interleaving.) There are two types of inconsistencies which he might find: (1) The observer sees the effects of concurrency. For example, action A spawns child A1 to read x (no update), and finds $x = 1$. Then A spawns child A2 (sequentially following A1) to read x, and A2 returns $x = 2$. (A has no other children.) This situation is clearly inconsistent with serializability. (2) The observer might deduce that the system state violates an invariant. For example, an observer at action A2 in Fig. 1.1 would see $x \neq y$.

The first type of inconsistency can be prevented by requiring that the information tree be serializable. Serializability of the information tree is too strong a condition, however, because the effects of other actions might be visible (through data objects) even though these actions are not in the information tree.

Since we want to formulate a consistency condition which does not depend on particular invariants for particular applications, we will increase the amount of information we presume is available to an observer. In other words, we will provide a *sufficient* consistency condition, which might not be necessary to insure consistency in all cases. We now assume that an observer at an action has complete knowledge of the set of possible behaviors of all other actions in the system (when run in isolation and to completion). We might imagine that the observer is given program listings for all actions, for example. This knowledge is sufficient to determine any invariants. (In a sense invariants are just one way of specifying certain aspects of program behavior.) Other than the actions in his information tree, he does not know what particular actions have actually run in the current execution, but if he is told that a particular action did run he can deduce the possible effects that it had (by checking his program listings).

The observer's view is *consistent* if he can *explain* the values in his information tree with a serial execution that conforms to the known behaviors of all actions. We stress again that the observer does not *know* what actions have run, but he can construct hypothetical execution histories based on his program listings. This condition is existential: an information tree is consistent if there exists a serializable "view tree" which contains the information tree and agrees with known behaviors of actions.

The problem with a condition defined in terms of program behaviors is that the *transaction system* does not have the program listings available to it (in a useful form). We imagine now that a "transaction manager" is placed at an action, and given its information tree. The transaction manager must decide whether the information tree is "consistent." The transaction manager will design algorithms to insure that an observer does not see an inconsistent state, but the manager does not have access to the program listings. But the transaction manager can devise a *sufficient* test for consistency: Since every action must run according to its program, the *actual behavior* of any action in the current execution must be among the allowed behaviors. Thus the transaction manager will try to create a "view tree" by taking actions from the real global action tree. (Of course, the observer cannot see this global tree.) Another way of looking at this restriction is to imagine that the program listings given to the observer are modified so that the only possible behavior of an action is the behavior it exhibited in the current execution.

The known behaviors of an action might include aborted actions as well as committed actions. For example, action B might run child B2 sequentially after child B1 in every execution. If B2 runs it can conclude that B1 has either committed or aborted. Moreover, if B commits, any other action can conclude that B1 committed or aborted, and that B2 committed or aborted. Note that if B aborted, then another action cannot conclude anything about B1 or B2 (since they might never have run at all).

Strictly speaking, the transaction manager should include these known aborts in its view trees, because they are part of "behavior." Just as we argued that there is no need to include these aborts in information trees, we can argue that there is no need to include them in view trees: Since aborted actions provide no information about their proper descendants, these proper descendants need not be included in the view tree. But aborted actions without descendants cannot affect serializability (by Lemma 2.6.2), so it is sufficient for the transaction manager to test for a serializable view tree which does not include these known aborts. It suffices for the transaction manager to choose actions for the view tree which are *visible* to A. (In other words, if a serializable view tree exists which includes these aborted actions, then it will still be serializable when the aborted actions are deleted. Thus we lose no generality by considering only view trees which do not contain these aborted actions.)

We place two restrictions on the selection of actions for this hypothetical view tree. First, the transaction manager must choose actions that are visible to the action whose tree he is constructing.

Second, because the behavior of an action might depend on any information available to it, if the transaction manager includes any action in his view tree, he must include the entire information tree of that action.

Example: We consider again the scenario presented in Fig. 1.1. Suppose that except for action A, the top level actions in the system each create two (sequentially related) subactions; the first subaction reads and increments x, and the second subaction reads and increments y. (Action A simply reads x and then reads y.) The initial values of x and y are 0. The information tree for A2 from the tree in Fig. 1.1 indicates that $x = 0, y = 1$. If the transaction manager were allowed to create a view tree which included only part of action B (i.e. included descendant B2 but excluded B1), he would conclude (wrongly) that A2's view is consistent.

Note also that the status of proper ancestors of the action should be 'active', since the observer should be able to believe that its proper ancestors are active (though in fact they might have committed or aborted). We include these proper ancestor in the view tree, but we exclude them from the information set closure requirement because (as discussed in the section on information trees) we have short-circuited these ancestors with our definition of information flow. (Thus we require the vertices of the view tree to be info-set$_T$//A-closed, rather than info-set$_T$-closed.)

For convenience, we separate the serializability requirement from the other requirements, and we define a view tree as any tree which satisfies the proper closure properties.

Defn 3.3.1: Let T be an action tree, $A \in$ vertices$_T$. Let $S = (T\lceil V)//A$, for some set $V \subseteq$ vertices$_T$. We say S is a view tree for A in T iff

1. $A \in V$

2. V is anc-closed

3. V is info-set$_T$//A-closed

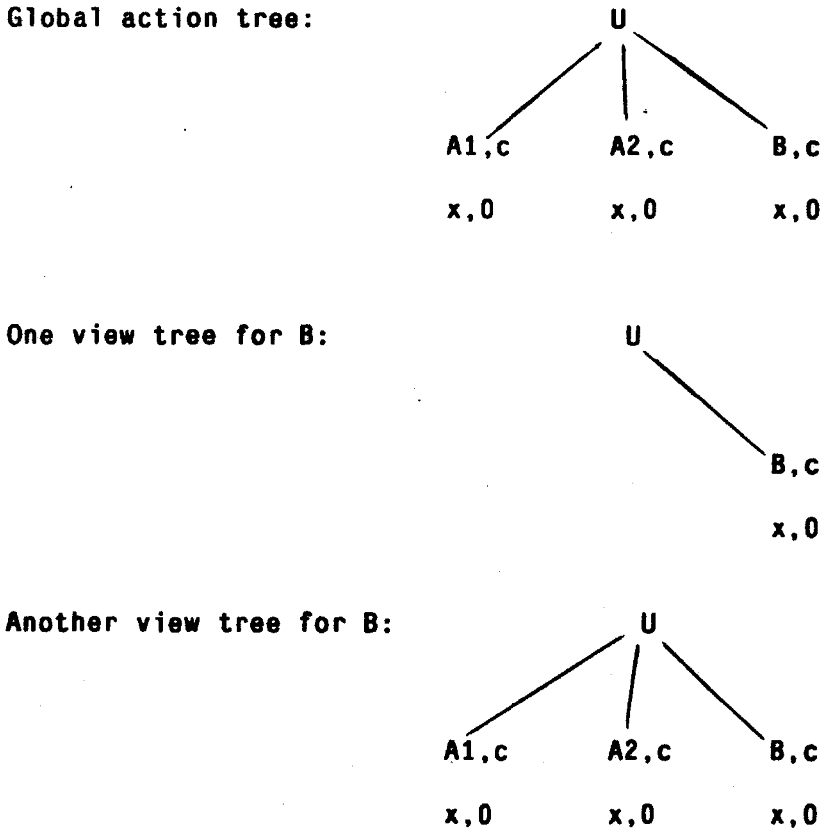
4. $V \subseteq$ visible$_T$(A)

(Note that $A \in V$ and V is info-set$_T$//A-closed $\Rightarrow$ info-tree$_T$(A) is a restriction of S.)

It is important to stress again that there is not enough information in action trees alone to determine the view tree for an action: a view tree is one of possibly several explanations for an information tree. As a trivial example, suppose that actions A1, A2, and B read object x in this order, but never update it. (See Fig. 3.2.) Then any combination of actions that includes B forms a serializable view tree for B.

For AATs, we will define a *particular* view tree, using the data ordering. To conclude that this is necessarily the view tree is incorrect: use of this particular view tree requires assumptions about how versions of objects are modified. We will use this view tree for one of our system models, but again note that the *definition* of a view tree is independent from the construction of this particular view tree.

Fig. 3.2. Multiple View Trees



3.4 View-Serializability

The view of an action is "consistent" if it can be explained with a *serializable* view tree. An entire tree is "view-serializable" if every action has a serializable view tree.

Defn 3.4.1: Let T be an action tree. We say T is <u>view-serializable</u> provided the following is true: For each $A \in \text{vertices}_T$, there exists a serializable view tree for A in T.

View-serializability is our basic correctness condition modeling internal consistency. In fact view-serializability is a strong enough condition to model external consistency as well: The following lemma shows that perm(T) is the only possible view tree of the (virtual) top-action, U.

Lemma 3.4.2: Let T be an action tree. Then S is a view tree for U in T if and only if S = perm(T).

Proof: Suppose that $S = (T|V)//U$ is a view tree for U in T; we show that S = perm(T). But $\text{prop-anc}(U) = \emptyset \Rightarrow S = T|V$.

Since V is $\text{info-set}_T//U$-closed, V is info-set_T-closed (again, because $\text{prop-anc}(U) = \emptyset$).

Thus V is v-child_T-closed, $\Rightarrow \text{v-desc}_T(U) \subseteq V$ (since $U \in V$).

But $\text{v-desc}_T(U) = \text{visible}_T(U) \Rightarrow \text{visible}_T(U) \subseteq V$.

But $V \subseteq \text{visible}_T(U)$, since S is a view tree for U.

Thus $V = \text{visible}_T(U)$, $\Rightarrow S = T|\text{visible}_T(U) = \text{perm}(T)$.

Conversely, let S = perm(T); we show S is a view tree for U in T. As above, $S = (T|\text{visible}_T(U))//U$. Let $W = \text{visible}_T(U)$. We show that W satisfies the correct closure properties for view trees.

1. $U \in W$, since $U \in \text{visible}_T(U)$.

2. If $A \in W$, then $\text{anc}(A) - \{U\} \subseteq \text{committed}_T$. If $B \in \text{anc}(A) - \{U\}$, then $\text{anc}(B) - \{U\} \subseteq \text{committed}_T$, $\Rightarrow B \in W$. If B = U, then $B \in W$ by (1) above. Thus W is anc-closed.

3. We show that W is $\text{info-set}_T//U$-closed, i.e. if $A \in W - \{U\}$, and $B \in \text{info-set}_T(A)$, then $B \in W$. But $A \in W - \{U\} \Rightarrow A \in \text{visible}_T(U)$, by definition. $B \in \text{info-set}_T(A) \Rightarrow B \in \text{visible}_T(A)$, by Lemma 3.2.5.5. Thus $B \in \text{visible}_T(U)$ by Lemma 2.2.3.1c, $\Rightarrow B \in W$.

4. $W \subseteq \text{visible}_T(U)$ by definition. ∎

Thus view-serializability implies serializability of perm(T); our condition for external consistency is covered by our condition for internal consistency.

Lemma 3.4.3: Let T be an action tree, then

T is view-serializable $\Rightarrow$ perm(T) is serializable.

Proof: Immediate from Lemma 3.4.2. ∎

3.5 Augmented Action Trees and Data-closed View Trees

We extend all definitions and lemmas for information sets, information trees, view trees, and view-serializability to AAT's in the obvious way (by applying them to erase(T)). (There is a subtle point that the definition of *restriction* of an AAT is different from the definition for an action tree, since a restriction of an AAT includes the data ordering from the original AAT. But the data ordering does not enter into any of the preceding definitions or lemmas, and $\text{erase}(T)|V = \text{erase}(T|V)$ for all AAT's, T, and action sets, V.)

For AAT's we define a particular view tree by augmenting the information tree via a type of data-closure. For the models that we will consider (in which only explicit aborts are allowed, and versions of objects change only in response to explicit commits and aborts), this view tree will be used to show view-serializability.

Defn 3.5.1: Let T be an AAT, $A \in \text{vertices}_T$. Define $\underline{\text{vtree}}_T(A)$ as follows:

Let $V = \text{vset}_T(A)$ ($= \text{v-precedes}^*_T(A)$)

The components of S are as follows:

-- $\text{vertices}_S = V \cup \text{prop-anc}(A)$

-- status_S is defined by

1. $B \in V - \{A\} \Rightarrow \text{status}_S(B) = \text{'committed'}$

2. $\text{status}_S(A) = \text{status}_T(A)$

3. $A' \in \text{prop-anc}(A) - V \Rightarrow \text{status}_S(A') = \text{'active'}$

-- If $B \in \text{datasteps}_S$, then $\text{label}_S(B) = \text{label}_T(B)$.

-- $\text{data}_S = \text{data}_T \cap \text{vertices}_S{}^2$

Unlike the situation for information sets, the view set of an action might include proper ancestors of that action. (This case occurs only when the view set "cycles back" to ancestors of the action; proper ancestors are not originally included in the view set.) The following lemma shows that $\text{vtree}_T(A)$ is a view tree for A if these cycles do not occur:

Lemma 3.5.2: Let T be an AAT, $A \in \text{vertices}_T$, $S = \text{vtree}_T(A)$. If $\text{prop-anc}(A) \cap \text{vset}_T(A) = \emptyset$, then S is a view tree for A in T.

Proof: Let $V = \text{vset}_T(A)$, $W = V \cup \text{prop-anc}(A)$. First we show that $S = (T|W)//A$.

By definition, $\text{vertices}_S = V \cup \text{prop-anc}(A) = W$. If $B \in V - \{A\}$, then $\text{status}_S(B) =$ 'committed'. But by Lemma 2.3.2, $\text{status}_T(B) =$ 'committed'. For $B \in \text{prop-anc}(A) - V$, $\text{status}_S(B) =$ 'active', by definition. But $\text{prop-anc}(A) \cap V = \emptyset \Rightarrow \text{prop-anc}(A) - V = \text{prop-anc}(A)$. Thus $B \in \text{prop-anc}(A) \Rightarrow \text{status}_S(B) =$ 'active'. For action A, $\text{status}_S(A) = \text{status}_T(A)$, by definition. Thus the trees S and $(T|W)//A$ agree on the status of all actions.

It is trivial to verify that these trees agree on all labels, and on the data ordering.
Now we show that W satisfies the correct closure properties for view trees:

1. $A \in W$ by definition

2. W is anc-closed by Lemma 2.3.4.

3. We show that W is $\text{info-set}_T//A$-closed, i.e. that $(\text{info-set}_T//A)(W) \subseteq W$. But by definition, $\text{info-set}_T//A$ is $\emptyset$ on prop-anc(A), and is identical to info-set_T otherwise. Thus we must show $\text{info-set}_T(V) \subseteq W$.

   But $V = \text{vset}_T(A) \Rightarrow$ V is vset_T-closed by Lemma 2.3.3a.

   $\text{vset}_T = \text{vprecedes}_T^* = (\text{v-anc-seq}_T \cup \text{v-child}_T \cup \text{v-data-anc}_T)^*$. But
   $\text{info-set}_T = (\text{v-anc-seq}_T \cup \text{v-child}_T)^*$.
   Thus $\text{info-set}_T(V) \subseteq \text{vset}_T(V)$.
   Thus V is vset_T-closed $\Rightarrow$ $\text{vset}_T(V) \subseteq V$, $\Rightarrow$ $\text{info-set}_T(V) \subseteq V$, $\Rightarrow$

info-set$_T$(V) $\subseteq$ W.

4. V $\subseteq$ visible$_T$(A), prop-anc(A) $\subseteq$ visible$_T$(A),
$\Rightarrow$ W $\subseteq$ visible$_T$(A). ▮

We will show in Chapter 6 that these cycles can only occur for view sets of orphans, and that the orphan detection strategy which we present will eliminate these cycles.

As examples of the construction of these data-closed view trees, vtree$_T$(A2) for the tree of Fig. 1.1 is the entire tree, and it is not serializable. For the tree of Fig. 3.2, vtree$_T$(B) is also the entire tree, but it is serializable.

4. Event-State Algebras

This chapter defines our basic execution model: the event-state algebra. An event-state algebra is a state-transition model of a system where events can occur asynchronously. A correctness proof for an event-state algebra shows that the states generated by valid event sequences satisfy some property. A strategy of hierarchical correctness proofs is explained: We define mappings between event-state algebras, and we give conditions on these mappings which insure that they preserve validity of event sequences. Finally, we present a model for distributed systems which is a special case of event-state algebras.

4.1 Event Algebras

4.1.1 Notation

If S is a (finite or infinite) set of symbols, then S^* denotes the set of finite sequences of symbols from S, including Λ -- the empty sequence. We will often drop the distinction between a symbol and a sequence of length one.

$\mathcal{N}$ denotes the set of non-negative integers, and $|u| \in \mathcal{N}$ denotes the length of sequence u.

If sequence u is a prefix of sequence v, then we write $u \leq v$. (Context will dictate whether "$\leq$" refers to the prefix relation on sequences or to numerical order on integers.) We say a set of sequences, W, is <u>prefix-closed</u> if and only if all prefixes of every sequence in W are also in W: $(\forall v \in W)(u \leq v \Rightarrow u \in W)$.

If $u \in S^*$ is a sequence, and $e \in S$, then we write $e \in u$ iff e is among the elements of u. (Note that, *a priori*, e might be repeated in u many times.) We denote by $\underset{u}{\rightarrow}$ the ordering on elements of u, i.e. if e,f $\in$ S, then

$$e \underset{u}{\rightarrow} f \Leftrightarrow u = a{\bullet}e{\bullet}b{\bullet}f{\bullet}c, \text{ for some sequences } a,b,c \in S^*.$$

Note that $\underset{u}{\rightarrow}$ is transitive for any $u \in S^*$. It is not necessarily acyclic, since elements of a sequence can be repeated.

4.1.2 Events and Valid Execution Sequences

An event algebra is a behavioral model of a system which describes the events in the system, and some constraints on "valid" executions imposed by the system. An execution of a system is any sequence of events from the system; the valid execution sequences will be some subset of these sequences. This type of model is useful for describing systems where events occur asynchronously and independently (as opposed to a program model, for example, where the allowable sequences of events are governed by a (generally sequential) program). It is also useful for describing properties of sequential systems which do not depend on the order of events (or depend on weaker ordering constraints than those enforced by the system).

At this level of description, "events" are completely uninterpreted: they should be regarded as textual symbols only. The only structure imposed by an event algebra is the set of valid execution sequences.

Defn 4.1.2.1: An event algebra is a pair

$$\mathcal{A} = (\mathcal{E}, \mathcal{V})$$

where $\mathcal{E}$ is a set (called the events of $\mathcal{A}$), and
$\mathcal{V}$ is a prefix-closed subset of $\mathcal{E}^*$ (called the valid execution sequences of $\mathcal{A}$).

(We will generally use symbols "e,f,g" to refer to individual events, and "u,v,w" to refer to sequences of events.)

We can consider the general problem faced in reasoning about a system to be showing some properties of the valid execution sequences. We are not interested (at this level) in *how* the system enforces the constraints on execution sequences. The valid execution sequences are simply a specification of the "correct" behaviors of the system.

4.1.3 Interpretations

We would often like to view a system at a higher level of abstraction than the one at which it is defined. In this section we describe an abstraction process for event algebras, and we show how this process can be used to organize proofs of system properties.

Defn 4.1.3.1: Let $\mathcal{A}_1 = (\mathcal{E}_1, \mathcal{T}_1)$, and $\mathcal{A}_2 = (\mathcal{E}_2, \mathcal{T}_2)$ be event algebras. An interpretation from $\mathcal{A}_1$ to $\mathcal{A}_2$ is a mapping $h: \mathcal{E}_1^* \rightarrow \mathcal{E}_2^*$.

An interpretation, h, is valid iff $h(\mathcal{T}_1) \subseteq \mathcal{T}_2$.

Note that any event sequence in one algebra can be interpreted as any sequence in another algebra: there are no constraints on this mapping. Although most interpretations of interest will have more structure (for example, h might be monotonic), it is not necessary to introduce this structure for these general definitions.

In proving a property of valid execution sequences for some event algebra, it might be useful to state this property as a constraint on execution sequences of an event algebra which is at a higher level of abstraction than the low-level model of the system of interest. (We might be interested only in particular events, for example, or we might regard a sequence of events as a single event at a higher level.) We might also want to break this abstraction process into several steps, constructing event algebras at intermediate levels of abstraction. We must then define valid interpretations between successive levels. Soundness of this technique follows directly from the following lemma:

Lemma 4.1.3.2: Let $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ be event algebras. If g is a valid interpretation from $\mathcal{A}_1$ to $\mathcal{A}_2$, and h is a valid interpretation from $\mathcal{A}_2$ to $\mathcal{A}_3$, then $h \circ g$ is a valid interpretation from $\mathcal{A}_1$ to $\mathcal{A}_3$.

Proof: Straightforward. ▮

Of course, we must be careful in applying this technique to be sure that the composition of mappings from lower-level algebras to higher-level algebras is consistent with the abstraction we desire from the lowest-level event sequences to the "abstract" event sequences.

We can reduce any problem of proving a property of valid execution sequences to an equivalent problem of constructing a valid interpretation: Suppose $\mathcal{A}_1 = (\mathcal{E}_1, \mathcal{V}_1)$ is an event algebra, and $P \subseteq \mathcal{E}_1^*$ is some property of execution sequences. We want to show that P holds for all valid execution sequences in $\mathcal{A}_1$, i.e. that $\mathcal{V}_1 \subseteq P$. We can construct a "higher-level" algebra, $\mathcal{A}_0$, whose valid execution sequences are just those specified by P: $\mathcal{A}_0 = (\mathcal{E}_1, P)$. If we define interpretation h from $\mathcal{A}_1$ to $\mathcal{A}_0$ as the identity map on event sequences, then $\mathcal{V}_1 \subseteq P$ if and only if h is valid.

By defining a top-level event algebra whose valid execution sequences automatically satisfy a desired property, we create a very uniform structure for our proofs: A "correctness" proof consists of definitions for a sequence of algebras, definitions for interpretations between levels, and proofs that all interpretations are valid.

4.1.4 Event-Homomorphic Interpretations

We defined interpretations very generally as any mapping between event sequences. Usually natural interpretations will have more structure, which will simplify a proof of validity. We define here a class of interpretations called "event-homomorphic" which allow the interpretation of any sequence to be constructed inductively from an interpretation of each event in the sequence.

Defn 4.1.4.1: Let $\mathcal{A}_1 = (\mathcal{E}_1, \mathcal{V}_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \mathcal{V}_2)$ be event algebras, and $h: \mathcal{E}_1^* \rightarrow \mathcal{E}_2^*$ be an interpretation from $\mathcal{A}_1$ to $\mathcal{A}_2$. We say h is an <u>event-homomorphic</u> interpretation iff

$$\forall u,v \in \mathcal{E}_1^*,\ h(uv) = h(u)h(v)$$

(Note that if h is event-homomorphic, then $h(\Lambda) = h(\Lambda\Lambda) = h(\Lambda)h(\Lambda)$; thus $h(\Lambda) = \Lambda$.)

If an interpretation, h, is event-homomorphic, then the image of any sequence can be constructed from the images of each element in the sequence. Thus we can specify an event-homomorphic interpretation as a mapping $h: \mathcal{E}_1 \rightarrow \mathcal{E}_2^*$.

Note that for an event-homomorphic interpretation, individual events in the lower-level algebra can be interpreted as any sequence of events in the higher-level algebra. A lower-level event which maps to Λ is effectively "abstracted out" at the higher level. A lower-level event which maps to a single event is visible at the higher level, although different lower-level events might map to the same higher-level event. To model the usual notion of "abstraction," where several "concrete" events might implement a single "abstract" event, we could map the earlier steps of the concrete sequence to Λ, and map the last step to the abstract event.

Our notion of "abstraction" is unusual, however, in that the image of a *single* lower-level event might be *several* higher-level events. We allow this case because the "observer" of a system might not be able to see the granularity of events directly: he might only see their *effects* (e.g. through changes in "state" caused by events). An "abstraction" in this sense might be a higher-level way of *explaining* these effects. It is possible that higher-level events can be "simpler" to understand, even though they are less "powerful" in that several higher-level events are needed to explain a single lower-level event.

We will deal only with event-homomorphic interpretations; in the remainder of this paper, "interpretation" always means "event-homomorphic interpretation."

4.2 Event-State Algebras

4.2.1 Events as State Transitions

Although our notion of the behavior of a system depends only upon the events in the system and the valid execution sequences, it is often convenient to describe a system by referring to a "system state." Specifically, we can abstract from event sequences to "states" by interpreting events as operations on a state. We introduce a structure called an "event-state algebra," which includes state as a basic system component.

Following [Stark83], we regard the *events* in a system as the fundamental entities; we introduce states for *convenience* in specifying the valid event sequences. The concept of "state" allows us to describe valid event sequences *inductively* by giving "preconditions" on the current state for each event. Because it is often simpler to reason *incrementally* about system behavior, states are a useful specification device. From this perspective, a system could be described (equally well) by several event-state algebras using different state spaces; these different state spaces would simply represent different ways of summarizing execution histories.

Defn 4.2.1.1: An <u>event-state</u> <u>algebra</u> is a quadruple

$$\mathcal{A} = (\mathcal{E}, \Sigma, \sigma, \tau)$$

where $\mathcal{E}$ is a set of <u>events</u>,

Σ is the set of system states,

$\sigma \in \Sigma$ is the initial system state, and

$\tau \subseteq \mathcal{E} \times \Sigma \times \Sigma$ is the transition relation.

Let $\tau(e) = \{(s,t) \in \Sigma^2: (e,s,t) \in \tau\}$.

For convenience, we require that $\tau(e)$ be a partial function on Σ, i.e.

$(e,s,t1), (e,s,t2) \in \tau \Rightarrow t1 = t2$.

(We could allow $\tau(e)$ to be an arbitrary *relation*, modeling a nondeterministic choice of the "next state." Because we will not need this power, we restrict $\tau(e)$ to a partial function.)

We regard $\tau(e)$ as a total function on $\Sigma \cup \{\perp\}$ (where $\perp$ represents "undefined") by defining

$\tau(e)(\perp) = \perp$, and

$\tau(e)(s) = \perp$ for $s \in \Sigma$ if there is no pair $(s,t) \in \tau(e)$.

If $s \in \Sigma \cup \{\perp\}$ and $e \in \mathcal{E}$, then we write

se for $\tau(e)(s)$

We generally drop the distinction between the event e and the partial function $\tau(e)$ when the meaning is clear. We extend our notation to sequences of events in the obvious way:

$$(s)(e_1e_2...e_n) = (((s)e_1)e_2)...e_n)$$

If $u \in \mathcal{E}^*$, then we say σu is the result of execution sequence u. (Note that the result might be $\perp$.)

If $\exists u \in \mathcal{E}^*: s2 = (s1)u$, (for $s1, s2 \in \Sigma$) then we write $s1 \vdash s2$ in $\mathcal{A}$, and we say s2 is reachable from s1 in $\mathcal{A}$. We will simply write $s1 \vdash s2$ when the algebra is clear from context.

If $H \subseteq \mathcal{E}^*$, and $s \in \Sigma$, then we define

$$sH = \{su: u \in H\}$$

(Similarly if $S \subseteq \Sigma$ and $u \in \mathcal{E}^*$, we define Su, or if $S \subseteq \Sigma$ and $H \subseteq \mathcal{E}^*$, we define SH.)

$\mathcal{V}(\mathcal{A})$, the set of valid execution sequences of $\mathcal{A}$, consists of all sequences whose result is defined (i.e. each event in the sequence is defined on the result of the preceding sequence):

$$e \in \mathcal{V}(\mathcal{A}) \Leftrightarrow \sigma e \neq \perp$$

$\mathcal{R}(\mathcal{A})$, the set of reachable states in $\mathcal{A}$, is the set of all states that are reachable from the initial state:

$$\mathcal{R}(\mathcal{A}) = \{s \in \Sigma: \sigma \vdash s\} \quad (\text{Thus } \mathcal{R}(\mathcal{A}) = \sigma\mathcal{V}(\mathcal{A}).)$$

We extend this definition to sequences of reachable states as follows:

$$\mathcal{R}^{(n)}(\mathcal{A}) = \{\langle s_1, s_2, \ldots.., s_n\rangle \in \Sigma^n: \sigma \vdash s_1 \vdash s_2 \ldots \vdash s_n\}$$

Note that $\mathcal{R}^{(n)}(\mathcal{A}) \subseteq (\mathcal{R}(\mathcal{A}))^n$.

We will use boldface symbols to refer to vectors of states, e.g. $\mathbf{s} = \langle s_1, s_2, \ldots, s_n\rangle$.

We denote by $\mathrm{PRE}_{\mathcal{A}}(e)$ the proper domain of $\tau(e)$, for each $e \in \mathcal{E}$. ($\mathrm{PRE}_{\mathcal{A}}(e) = \{s \in \Sigma: \tau(e)(s) \neq \perp\}$.) We generally drop the subscript when the algebra is clear from context. We extend this notation to sequences $u \in \mathcal{E}^*$ by defining:

$$\mathrm{PRE}(u) = \{s \in \Sigma: su \neq \perp\}$$

(In general, if an event-state algebra is named "$\mathcal{A}_n$" for some subscript, "n", then we will abbreviate $\mathcal{V}(\mathcal{A}_n)$ as "$\mathcal{V}_n$", $\mathcal{R}(\mathcal{A}_n)$ as "$\mathcal{R}_n$", and $\mathrm{PRE}_{\mathcal{A}_n}$ as "PRE_n.")

We are viewing event-state algebras as convenient structures for specifying event algebras. We say that an event-state algebra $\mathcal{A}' = (\mathcal{E}', \Sigma', \sigma', \tau')$ is a presentation of event algebra $\mathcal{A} = (\mathcal{E}, \mathcal{V})$ if and only if $\mathcal{E}' = \mathcal{E}$ and $\mathcal{V}(\mathcal{A}') = \mathcal{V}$. (Note that $\mathcal{V}(\mathcal{A}')$ must be prefix closed by construction.) It follows from this definition that several event-state algebra presentations might exist for a given event algebra, but each event-state algebra is a presentation of a unique event algebra. If $\mathcal{A}'$ is a presentation of $\mathcal{A}$, then we say that $\mathcal{A}$ is the embedded event algebra for $\mathcal{A}'$.

We can show that an event-state algebra presentation exists for any event-algebra -- the degenerate presentation whose state is the entire execution history:

Lemma 4.2.1.2: For any event algebra $\mathcal{A} = (\mathcal{E}, \mathcal{V})$, there exists an event-state algebra presentation of $\mathcal{A}$.

Proof: Let $\mathcal{A}' = (\mathcal{E}', \Sigma', \sigma', \tau')$, where
$\mathcal{E}' = \mathcal{E}$, $\Sigma' = \mathcal{E}^*$, $\sigma' = \Lambda$, and $\tau' = \{(e,u,ue)\text{: } e \in \mathcal{E},\ ue \in \mathcal{V}\}$. Then $\mathcal{V}(\mathcal{A}') = \mathcal{V}$, so $\mathcal{A}'$ is a presentation of $\mathcal{A}$. ∎

Thus we will deal only with event-state algebras from here, with no loss in generality.

An interpretation from one event-state algebra to another is defined to be any interpretation between the embedded event algebras. This interpretation is valid if and only if the interpretation between embedded event algebras is valid.

4.2.2 Possibilities Maps

Because we are using states to describe the valid execution sequences of an event-state algebra, it is natural to use these states in proving that an interpretation between event-state algebras is valid. Capturing execution histories with states allows us to specify valid execution sequences *inductively*; by extending the event mapping of an interpretation to a mapping between state sets, we will give an inductive technique for proving that the interpretation is valid.

The state mappings we will define are somewhat unusual in that we allow a mapping from states at the lower level to *sets of states* at the higher level. We call these mappings possibilities maps (if they satisfy certain properties), because they give a set of possible higher-level states which correspond to each lower-level state. Because the states in an event-state algebra can represent any convenient summary of execution histories, it is possible that the higher-level state might retain *more* information about executions than the lower-level state. In this case there is not enough information in the lower-level state to uniquely determine the higher-level state. Thus we permit "looser" mappings which specify the set of states which are consistent with (are "possibilities" for) a given lower-level state.

Possibilities maps are particularly useful when the lower-level state is *distributed*, and the higher-level algebra is a *global interpretation* of the lower-level algebra. (It might be convenient to specify a distributed algorithm in terms of a "virtual" global state, for example.) Because the lower-level state is partitioned among components, each component has only partial knowledge of the total system state. Thus there will generally be several higher-level states which are "possibilities" given the state at an individual component. This "partial information" property of distributed systems makes possibilities maps a natural tool for describing interpretations of these systems.

Possibilities maps can be regarded as a generalization of the standard notion of *homomorphism.* The state mapping of a homomorphism is a single-valued function, because the higher-level state space is always "more abstract" (has *less* detailed information) than the lower-level state space.

If $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ are event-state algebras, then we will write $h\colon \mathcal{A}_1 \rightarrow \mathcal{A}_2$ if $h\colon \mathcal{E}_1 \rightarrow \mathcal{E}_2^*$ and $h\colon \Sigma_1 \rightarrow \mathcal{P}(\Sigma_2)$. (We use "h" to denote both the event mapping and the state mapping; the meaning will be clear from context.) Note that $h\colon \mathcal{A}_1 \rightarrow \mathcal{A}_2$ does *not* necessarily imply that h satisfies any special properties; in particular, h need not be a possibilities map. We say that the proper domain of h (of the state mapping) is: $\mathrm{domain}(h) = \{s \in \Sigma_1\colon h(s) \neq \emptyset\}$.

We extend a mapping $h\colon \Sigma_1 \rightarrow \mathcal{P}(\Sigma_2)$ to a mapping $h\colon \Sigma_1^n \rightarrow \mathcal{P}(\Sigma_2^n)$ by defining $h(\langle s_1,s_2,\ldots,s_n\rangle) = \{\langle t_1,t_2,\ldots, t_n\rangle\colon t_i \in h(s_i), \text{ for } i = 1,2,\ldots,n\}$.

Defn 4.2.2.1: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let $h\colon \mathcal{A}_1 \rightarrow \mathcal{A}_2$. We say h is a possibilities map iff

1. h preserves initial states:

   $\sigma_2 \in h(\sigma_1)$

2. h preserves events

   $s \in \mathrm{PRE}_1(e) \cap \mathcal{R}_1,\ t \in h(s) \cap \mathcal{R}_2$
   $\Rightarrow (t)h(e) \in h(se)$

   (Note that $(t)h(e) \in h(se) \Rightarrow (t)h(e) \neq \perp$, since $h(se) \subseteq \Sigma_2$.)

In many cases we will not need the full power of possibilities maps to map from states to sets of states. If a mapping $h\colon \mathcal{A}_1 \rightarrow \mathcal{A}_2$ has the property that $\forall s \in \Sigma_1,\ |h(s)| \leq 1$, then we will consider h to be a partial mapping from Σ_1 to Σ_2, and we will change our notation accordingly. (For example, we will write $t = h(s)$ instead of $t \in h(s)$.)

We will use the properties of possibilities maps to prove inductively that a mapping is valid. As an intermediate step, we define the notion of a *faithful* mapping. We then show the main result for possibilities maps: any possibilities map is a valid interpretation.

Defn 4.2.2.2: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let $h: \mathcal{A}_1 \rightarrow \mathcal{A}_2$. For $k \in \mathcal{N}$, we say that h is k-faithful iff $(\forall v \in \mathcal{V}_1: |v| \leq k)$, $\sigma_2 h(v) \in h(\sigma_1 v)$. We say h is faithful iff h is k-faithful for all $k \in \mathcal{N}$. Note that h preserves initial states if and only if h is 0-faithful.

Lemma 4.2.2.3: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let $h: \mathcal{A}_1 \rightarrow \mathcal{A}_2$. Then h is faithful $\Rightarrow$ h is a valid interpretation.

Proof: h is faithful $\Rightarrow \sigma_2 h(v) \in h(\sigma_1 v)\ \forall v \in \mathcal{V}_1$. Thus $\sigma_2 h(v) \neq \perp \Rightarrow h(v) \in \mathcal{V}_2$. ■

Lemma 4.2.2.4: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let $h: \mathcal{A}_1 \rightarrow \mathcal{A}_2$. Then h is a possibilities map $\Rightarrow$ h is faithful.

Proof: Suppose h is a possibilities map. Then h preserves initial states $\Rightarrow$ h is 0-faithful. We show h is k-faithful $\Rightarrow$ h is k+1-faithful.

Let $ve \in \mathcal{V}_1$, $|v| = k$, $e \in \mathcal{E}_1$. Since h is k-faithful, $\sigma_2 h(v) \in h(\sigma_1 v)$. But $ve \in \mathcal{V}_1 \Rightarrow \sigma_1 v \in PRE_1(e) \cap \mathfrak{R}_1$. And $\sigma_2 h(v) \in h(\sigma_1 v) \cap \mathfrak{R}_2$. Since h preserves events, $\sigma_2 h(v) h(e) \in h(\sigma_1 ve)$, $\Rightarrow$ h is k+1-faithful. ■

Lemma 4.2.2.5: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let $h: \mathcal{A}_1 \rightarrow \mathcal{A}_2$. Then h is a possibilities map $\Rightarrow$ h is a valid interpretation.

Proof: Immediate corollary of Lemmas 4.2.2.4 and 4.2.2.3. ■

We will often find it useful to prove preservation of events in two parts: We will assume that preconditions are satisfied and show that transitions behave correctly under the interpretation; we will show separately that preconditions are satisfied:

Lemma 4.2.2.6: Let $\mathcal{A}_1 = (\mathcal{S}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{S}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let $h: \mathcal{A}_1 \rightarrow \mathcal{A}_2$. Then h preserves events if and only if

1. <u>h preserves transitions:</u>

   $s \in PRE_1(e) \cap \mathfrak{R}_1,\ t \in h(s) \cap PRE_2(h(e)) \cap \mathfrak{R}_2$
   $\Rightarrow (t)h(e) \in h(se)$

2. <u>h preserves preconditions:</u>

   $s \in PRE_1(e) \cap \mathfrak{R}_1,\ t \in h(s) \cap \mathfrak{R}_2$
   $\Rightarrow t \in PRE_2(h(e))$

Proof: Suppose h preserves events. Then

$s \in PRE_1(e) \cap \mathfrak{R}_1,\ t \in h(s) \cap \mathfrak{R}_2,$
$\Rightarrow (t)h(e) \in h(se),$
$\Rightarrow (t)h(e) \neq \perp,$
$\Rightarrow t \in PRE_2(h(e)),$ so h preserves preconditions.

$s \in PRE_1(e) \cap \mathfrak{R}_1,\ t \in h(s) \cap PRE_2(h(e)) \cap \mathfrak{R}_2,$
$\Rightarrow\ s \in PRE_1(e) \cap \mathfrak{R}_1,\ t \in h(s) \cap \mathfrak{R}_2,$
$\Rightarrow (t)h(e) \in h(se),$ so h preserves transitions.

Conversely, suppose h preserves preconditions and transitions. Then

$s \in PRE_1(e) \cap \mathfrak{R}_1,\ t \in h(s) \cap \mathfrak{R}_2,$
$\Rightarrow t \in PRE_2(h(e)),$ since h preserves preconditions.

Thus $s \in PRE_1(e) \cap \mathfrak{R}_1,\ t \in h(s) \cap PRE_2(h(e)) \cap \mathfrak{R}_2,$
$\Rightarrow (t)h(e) \in h(se),$ since h preserves transitions. ∎

4.2.3 Canonical Possibilities Map

We can show that the method of constructing a possibilities map between event-state algebras is a completely general technique for proving validity: Given any (event-homomorphic) valid interpretation, an extension of this interpretation to a possibilities map always exists:

Lemma 4.2.3.1: Let $\mathcal{A}_1 = (\mathcal{S}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{S}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and $h: \mathcal{E}_1 \rightarrow \mathcal{E}_2^*$ be a valid interpretation from $\mathcal{A}_1$ to $\mathcal{A}_2$. Then if we extend h to a state-set mapping as follows:

$$h(s) = \{\sigma_2 h(u): \sigma_1 u = s\}$$

then h is a possibilities map from $\mathcal{A}_1$ to $\mathcal{A}_2$.

Proof: First we show that $h(s) \subseteq \Sigma_2$ for all $s \in \Sigma_1$ (i.e. $\perp \neq h(s)$): $\sigma_1 u = s \Rightarrow u \in \mathcal{T}_1$ $\Rightarrow h(u) \in \mathcal{T}_2$ (since h is valid) $\Rightarrow \sigma_2 h(u) \neq \perp$. Thus h does define a mapping from Σ_1 to $\mathfrak{P}(\Sigma_2)$.

Now we show that h satisfies the conditions for a possibilities map:

1. h preserves initial states:

   $\sigma_2 = \sigma_2 \Lambda = \sigma_2 h(\Lambda)$ (since h is event-homomorphic);
   $\sigma_2 h(\Lambda) \in \{\sigma_2 h(u): \sigma_1 u = \sigma_1\} = h(\sigma_1)$.

2. h preserves events:

   $s \in PRE_1(e) \cap \mathfrak{R}_1$, $t \in h(s) \cap \mathfrak{R}_2$.
   Let $s = \sigma_1 v$, $v \in \mathcal{T}_1$.
   So $t \in h(s) = \{\sigma_2 h(u): \sigma_1 u = \sigma_1 v\}$,
   $\Rightarrow t = \sigma_2 h(u)$ for some u: $\sigma_1 u = \sigma_1 v$.

   Now $(t)h(e) = \sigma_2 h(u) h(e) = \sigma_2 h(ue)$
   $\in \{\sigma_2 h(w): \sigma_1 w = \sigma_1 ve\} = h(se)$. ∎

Note that the set $h(s) = \{\sigma_2 h(u): \sigma_1 u = s\}$ corresponds intuitively to the "possibilities" for higher-level states associated with lower-level state s: The sequences $\{u: \sigma_1 u = s\}$ are the possible histories which might have generated state s; $\sigma_2 h(u)$ is the higher-level state that would have resulted from execution of u. Thus if we only know state s, then we can only "pin down" the possible higher-level state to the set $\{\sigma_2 h(u): \sigma_1 u = s\}$.

4.2.4 Invariants

We have reduced the task of showing that interpretation is valid to the task of proving that a mapping (on both states and events) is a possibilities map. It will often be convenient to use properties of reachable states (at both the higher and lower levels) in showing that a mapping is a possibilities map. We generalize the notion of an *invariant* to include properties of sequences of states as well as properties of single states. We also describe properties of individual components of the state, since we will show below that if a component is preserved by a state mapping between algebras, then in some cases we can carry invariants proved at the higher level for this component downward to the lower level (without re-proving the invariants at the lower level). Our development of an event-state algebra hierarchy for a transaction system will make extensive use of this method of carrying invariants down from higher level algebras.

4.2.4.1 Basic Definitions

Defn 4.2.4.1.1: Let $\mathcal{A} = (\mathcal{E}, \Sigma, \sigma, \tau)$ be an event-state algebra. If $I \subseteq \Sigma^n$, we say that I is an n-ary property in $\mathcal{A}$. If $n = 1$, then we will simply say "I is a property," and if $n = 2$, we will say "I is a pair-property."

Defn 4.2.4.1.2: Let $\mathcal{A} = (\mathcal{E}, \Sigma, \sigma, \tau)$ be an event-state algebra, and let $k \in \mathcal{N}$. If I is an n-ary property in $\mathcal{A}$, we say that I is k-invariant in $\mathcal{A}$ iff the following is true: For all sequences $(v_1,v_2,...,v_n) \in \mathcal{V}^n$ such that $v_1 \leq v_2 \leq ... \leq v_n$, and $|v_n| \leq k$, we have $(\sigma v_1,\sigma v_2,...,\sigma v_n) \in I$. We say that I is invariant in $\mathcal{A}$ iff $\forall k \in \mathcal{N}$, I is k-invariant in $\mathcal{A}$. Thus I is invariant in $\mathcal{A}$ iff $\mathcal{R}^{(n)}(\mathcal{A}) \subseteq I$.

We will usually drop the qualification "in $\mathcal{A}$" when the algebra is clear from context. Note that the case $n = 1$ corresponds to the usual notion of an "invariant." When we say that "I is *an* invariant," we will generally mean that I is a 1-ary property which is invariant. Similarly we will say "J is a pair-invariant" if J is a pair-property which is invariant.

4.2.4.2 Relative Invariants and Relative Possiblities Maps

To prove that a particular mapping is a possibilities map, we will frequently prove first some useful invariants for the higher and lower-level algebras. If we organize a proof hierarchically (with several levels of event-state algebras), we might find that we need the same invariants at several of these levels. While we could prove the needed invariants independently at each level, to do so might repeat a lot of work unnecessarily. Since faithful mappings map reachable states into reachable states, it might be easy to infer that higher-level invariants hold at the lower level *if we knew that the mapping between algebras were faithful.* In some cases, however, we might want to use these invariants to show that the mapping is a possibilities map (and hence is faithful, by Lemma 4.2.2.4). In these cases we are faced with a mutual dependency between invariants and a possibilities mapping.

Our solution to this mutual dependency depends on the fact that both invariants and possibilities maps are generally proved *inductively.* Conceptually, then, we will prove both an invariant and the possibilities map together with the same induction. For convenience, we separate the dependencies in our *definitions*; we define an invariant *relative to* a mapping, and a possibilities map *relative to* a property. Because the key property of possibilities maps is *faithfulness,* we also define faithfulness *relative to* a property, and we prove a lemma which is the "relative" version of Lemma 4.2.2.4. We also state a "relative" version of Lemma 4.2.2.6.

Defn 4.2.4.2.1: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let h: $\mathcal{A}_1 \rightarrow \mathcal{A}_2$. Let $P \subseteq \Sigma_1$ be a property in $\mathcal{A}_1$.

We say h is a possibilities map relative to P iff

1. h preserves initial states ($\sigma_2 \in h(\sigma_1)$)
2. h preserves events relative to P:

   $s \in PRE_1(e) \cap \mathcal{R}_1 \cap P,\ t \in h(s) \cap \mathcal{R}_2$
   $\Rightarrow (t)h(e) \in h(se)$

Defn 4.2.4.2.2: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, $P \subseteq \Sigma_1$, and let h: $\mathcal{A}_1 \rightarrow \mathcal{A}_2$.

We say h is faithful relative to P iff

1. h is 0-faithful

2. $(\forall k \in \mathcal{N})$ h is k-faithful, and P is k-invariant $\Rightarrow$ h is k+1-faithful

Lemma 4.2.4.2.3: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, $P \subseteq \Sigma_1$, h: $\mathcal{A}_1 \rightarrow \mathcal{A}_2$. If h is a possibilities map relative to P, then h is faithful relative to P.

Proof: h preserves initial state, so h is 0-faithful. Now suppose h is k-faithful, and P is k-invariant, for some $k \in \mathcal{N}$. We show h is k+1-faithful.

Take $v \in \mathcal{T}_1$, $|v| \leq k+1$. We must show that $\sigma_2 h(v) \in h(\sigma_1 v)$. If $|v| \leq k$ then the result follows directly since h is k-faithful, so assume $|v| = k+1$. Let $v = ue$, for some $u \in \mathcal{T}_1$, $e \in \mathcal{E}_1$ $(|u| = k)$.

Since h is k-faithful, $\sigma_2 h(u) \in h(\sigma_1 u)$. But P is k-invariant, so $\sigma_1 u \in P$. Since $ue \in \mathcal{T}_1$, $\sigma_1 u \in PRE_1(e)$.

Thus $\sigma_1 u \in PRE_1(e) \cap \mathcal{R}_1 \cap P$, $\sigma_2 h(u) \in h(\sigma_1 u) \cap \mathcal{R}_2$,

$\Rightarrow$ $\sigma_2 h(u)h(e) \in h(\sigma_1 ue)$, since h is a possibilities map relative to P,

$\Rightarrow$ $\sigma_2 h(v) \in h(\sigma_1 v)$. ∎

Lemma 4.2.4.2.4: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let h: $\mathcal{A}_1 \rightarrow \mathcal{A}_2$. Then h preserves events relative to P if and only if

1. h preserves transitions relative to P:

   $s \in PRE_1(e) \cap \mathcal{R}_1 \cap P$, $t \in h(s) \cap PRE_2(h(e)) \cap \mathcal{R}_2$
   $\Rightarrow$ $(t)h(e) \in h(se)$

2. h preserves preconditions relative to P:

   $s \in PRE_1(e) \cap \mathcal{R}_1 \cap P$, $t \in h(s) \cap \mathcal{R}_2$
   $\Rightarrow$ $t \in PRE_2(h(e))$

Proof: Similar to the proof of Lemma 4.2.2.6. ∎

Defn 4.2.4.2.5: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, $P \subseteq \Sigma_1$, and let h: $\mathcal{A}_1 \rightarrow \mathcal{A}_2$.

We say P is invariant relative to h iff

1. P is 0-invariant
2. $(\forall k \in \mathcal{N})$ P is k-invariant, and h is k+1-faithful $\Rightarrow$ P is k+1-invariant

We now show that we can prove invariants and possibilities maps together with the same induction:

Lemma 4.2.4.2.6: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, $P \subseteq \Sigma_1$. Let h be a possibilities map from $\mathcal{A}_1$ to $\mathcal{A}_2$ relative to P, and let P be invariant in $\mathcal{A}_1$ relative to h. Then h is a possibilities map, and P is invariant in $\mathcal{A}_1$.

Proof: Since h is a possibilities map relative to P, h is faithful relative to P (by Lemma 4.2.4.2.3). We show inductively on k that P is k-invariant, and h is k-faithful. P is 0-invariant and h is 0-faithful by definition. Assume P is k-invariant, and h is k-faithful. Since h is a faithful relative to P, h is k+1-faithful. Since P is invariant relative to h, P is k+1-invariant.

Thus P is invariant. To see that h is a possibilities map, note that h preserves initial states by definition. h preserves events, because

$s \in PRE_1(e) \cap \mathcal{R}_1 \Rightarrow s \in P$ since P is invariant. ∎

4.2.4.3 Invariants on Fixed Subspaces

We have described a process for proving an invariant simultaneously with proving a possibilities map. In this section we show that this technique can be useful when a particular subspace of the lower-level state space is unchanged by the state mapping.

Defn 4.2.4.3.1: Let $\mathcal{A} = (\mathcal{E}, \Sigma, \sigma, \tau)$ be an event-state algebra, let $\mathcal{J}$ be some index set, and let Σ be the Cartesian product of component sets, Γ_N, for $N \in \mathcal{J}$. We say that N is the *name* of component Γ_N. We assume that each component has a unique name. (We will frequently denote a component by a variable name used for an *instance* of the component set. For example, if $\Sigma = \Gamma_1 \times \Gamma_2$, and we use $\langle A,B \rangle \in \Sigma$ to represent an instance of the state, then we will refer to the "A-component," or the "B-component.") Let $N_1,N_2 \ldots N_k$ be distinct names from $\mathcal{J}$. We say that $\Gamma_{N_1} \times \Gamma_{N_2} \ldots \times \Gamma_{N_k}$ is a subspace of Σ, with name $N = \langle N_1,N_2,\ldots,N_k \rangle$. (Note that each such composite name denotes a unique subspace.) If $s \in \Sigma$, then "s.N" denotes the projection of s onto the subspace named by N. If $s = \langle s_1,s_2,\ldots,s_n \rangle$ is a vector of states, then s.N is defined in the obvious way as $\langle s_1.N,s_2.N,\ldots,s_n.N \rangle$.

We extend the definitions of n-ary properties and invariants to properties which only depend on a particular subspace.

Defn 4.2.4.3.2: Let $\mathcal{A} = (\mathcal{E}, \Sigma, \sigma, \tau)$ be an event-state algebra, and let N name a subspace, Γ, of Σ. If $I \subseteq \Gamma^n$, we say that I is an n-ary property for N.

Let I be an n-ary property for N, and let $I' = \{s \in \Sigma^n: s.N \in I\}$. We say I is invariant for N in $\mathcal{A}$ iff I' is invariant in $\mathcal{A}$. If $k \in \mathcal{N}$, then we say I is k-invariant for N in $\mathcal{A}$ iff I' is k-invariant in $\mathcal{A}$.

Invariants for a subspace are of interest when the state mapping between two algebras *fixes* that subspace. We will show below that invariants for a fixed subspace can be "carried down" to the lower-level algebra.

Defn 4.2.4.3.3: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let $h: \mathcal{A}_1 \rightarrow \mathcal{A}_2$. Suppose that the state spaces of $\mathcal{A}_1$ and $\mathcal{A}_2$ both contain a subspace, Γ, with name N. We say that h fixes N iff for all $s \in \Sigma_1$, and for all $t \in h(s)$, t.N = s.N. (Thus h does not change the N-subspace of the state.) It is straightforward to show that if h fixes N, $s \in \Sigma_1^n$, and $t \in h(s)$, then t.N = s.N.

Now we show how we can carry higher level invariants for fixed subspaces down to the lower level. Because we might want to use these invariants in inductive proofs (in particular, as we explain below, in inductive proofs of other *relative invariants* for the lower level), we state this lemma in "parameterized" form (i.e. in terms of k-invariants and k-faithful mappings).

Lemma 4.2.4.3.4: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, let $k \in \mathcal{N}$, and let $h: \mathcal{A}_1 \rightarrow \mathcal{A}_2$ be k-faithful. Let N name subspace Γ in both Σ_1 and Σ_2, and suppose that h fixes N. If n-ary property $I \subseteq \Gamma^n$ is invariant for N in $\mathcal{A}_2$, then I is k-invariant for N in $\mathcal{A}_1$.

Proof: Let $I2 = \{t \in \Sigma_2^n: t.N \in I\}$, $I1 = \{s \in \Sigma_1^n: s.N \in I\}$. I is invariant for N in $\mathcal{A}_2$, so $\mathfrak{R}_2^{(n)} \subseteq I2$. We must show that that I1 is k-invariant in $\mathcal{A}_1$. Let $\langle v_1,v_2,...,v_n\rangle \in \mathcal{V}_1^n$, with $v_1 \leq v_2 \leq ... v_n$, and $|v_n| \leq k$; we show that $s = \langle\sigma_1 v_1,\sigma_1 v_2,...,\sigma_1 v_n\rangle \in I1$. Since h is k-faithful, $\sigma_2 h(v_i) \in h(\sigma_1 v_i)$ for i = 1,2..n. Let $t = \langle\sigma_2 h(v_1),\sigma_2 h(v_2),...,\sigma_2 h(v_n))\rangle$. Then $t \in \mathfrak{R}_2^{(n)}$, because each $\sigma_2 h(v_i) \in \mathfrak{R}_2$, and $h(v_1) \leq h(v_2) \leq ... h(v_n)$ since h is event-homomorphic.

Since $t \in \mathfrak{R}_2^{(n)}$, $t \in I2$; thus $t.N \in I$. But $t \in h(s)$, and h fixes N, so $t.N = s.N$. Thus $s.N \in I$, $\Rightarrow s \in I1$. ▮

Because a mapping which is a possibilities map is necessarily faithful, and hence k-faithful for all k, we have the following lemma:

Lemma 4.2.4.3.5: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let $h: \mathcal{A}_1 \rightarrow \mathcal{A}_2$ be a possibilities map. Let N name subspace Γ in both Σ_1 and Σ_2, and suppose that h fixes N. If n-ary property $I \subseteq \Gamma^n$ is invariant for N in $\mathcal{A}_2$, then I is invariant for N in $\mathcal{A}_1$.

Proof: Immediate corollary of Lemma 4.2.2.4 and Lemma 4.2.4.3.4. ▮

We showed in Lemma 4.2.4.2.6 that if h is a possibilities map from $\mathcal{A}_1$ to $\mathcal{A}_2$ relative to property P, and P is invariant for $\mathcal{A}_1$ relative to mapping h, then it follows that h is a possibilities map. Because of Lemma 4.2.4.3.4, we can use known invariants for fixed subspaces in $\mathcal{A}_2$ to prove that P is invariant relative to h. Note that in proving that P is invariant relative to h, we can assume that h is k+1-faithful (instead of simply k-faithful) when showing P is k+1-invariant. By Lemma 4.2.4.3.4, we can thus assume that invariants from $\mathcal{A}_2$ for fixed subspaces are k+1-invariant.

We will generally apply Lemma 4.2.4.3.4 to 1-ary or 2-ary invariants. We summarize the

(s.N, t.N) ∈ J

⇒ t ∈ P (by the Induction Hypothesis of the Lemma),

⇒ P is k+1-invariant. ▮

It is important to understand exactly what Lemma 4.2.4.3.6 says: We cannot assume that the higher-level invariants (I and J) are truly *invariant* in $\mathcal{A}_1$, but we can assume they are *k+1-invariant* for the induction step of showing P invariant. Because we construct the induction so that faithfulness of h stays "one step ahead" of invariance of P, we can assume both t.N ∈ I, and (s.N,t.N) ∈ J above. (If we only knew that h were k-faithful, instead of k+1-faithful, then we would only be able to assume s.N ∈ I.)

4.2.5 Augmentation Maps and Auxiliary State

The power of possibilities maps to map a single state into a set of states is useful when the lower-level algebra is somehow "more abstract" than the higher-level algebra. If the higher-level model retains more information about a system than a lower-level model, then the low-level state will not uniquely determine the high-level state. Another technique for showing a valid interpretation from one algebra to another is to augment the lower-level state with *auxiliary variables.* These variables are "virtual" components of the state, in that they do not enter into any preconditions for events, and the transition effects on other components of the state are not affected by the auxiliary variables.

Defn 4.2.5.1: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras. We say that $\mathcal{A}_2$ is an augmentation of $\mathcal{A}_1$ with auxiliary state Aux iff

1. $\mathcal{E}_2 = \mathcal{E}_1$
2. $\Sigma_2 = \Sigma_1 \times \mathrm{Aux}$
3. $\sigma_2 = (\sigma_1, a_0)$ for some $a_0 \in \mathrm{Aux}$
4. $\forall e \in \mathcal{E}_1$, $\mathrm{PRE}_2(e) = \mathrm{PRE}_1(e) \times \mathrm{Aux}$ (i.e. the auxiliary state enters into no preconditions)
5. $(s,a) \in \mathrm{PRE}_2(e) \Rightarrow \tau_2(e)(s,a) = (\tau_1(e)(s),a')$ for some $a' \in \mathrm{Aux}$ (i.e. the auxiliary state does not affect transitions)

If $\mathcal{A}_2$ is an augmentation of $\mathcal{A}_1$ with Aux, then we define the augmentation map, h: $\mathcal{A}_1 \rightarrow \mathcal{A}_2$, as follows:

$\forall e \in \mathcal{E}_1$, $h(e) = e$
$\forall s \in \Sigma_1$, $h(s) = \{s\} \times \text{Aux}$.

Lemma 4.2.5.2: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let $\mathcal{A}_2$ be an augmentation of $\mathcal{A}_1$ with auxiliary state Aux. Then Σ_1 is a subspace of Σ_2, and the augmentation map, h, fixes Σ_1.

Proof: Straightforward from the definition. ∎

The following lemma shows a relationship between the technique of using auxiliary state, and the technique of defining a possibilities map: every augmentation map is a possibilities map.

Lemma 4.2.5.3: Let $\mathcal{A}_1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$ and $\mathcal{A}_2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$ be event-state algebras, and let $\mathcal{A}_2$ be an augmentation of $\mathcal{A}_1$ with auxiliary state Aux. Then the augmentation map, h, is a possibilities map.

Proof:

1. $h(\sigma_1) = \{(\sigma_1,a): a \in \text{Aux}\}$,
   $\Rightarrow (\sigma_1,a_0) \in h(\sigma_1)$.

2. Let $s \in \text{PRE}_1(e) \cap \mathcal{R}_1$, $t \in h(s) \cap \mathcal{R}_2$,
   $\Rightarrow t = (s,a)$ for some $a \in \text{Aux}$.

   $s \in \text{PRE}_1(e) \Rightarrow (s,a) \in \text{PRE}_2(h(e)) = \text{PRE}_2(e)$,
   $\Rightarrow (t)h(e) = te = \tau_2(e)(t) = (\tau_1(e)(s),a')$ for some a'.
   But $h(se) = \{(se,a): a \in \text{Aux}\} = \{(\tau_1(e)(s),a): a \in \text{Aux}\}$,
   $\Rightarrow (t)h(e) \in h(se)$. ∎

4.3 Distributed System Model

We model a distributed system as a special type of event-state algebra. First we present a general framework for a "distributed algebra," and then we specialize further to a particular model for the distributed environment of our transaction system. While these models have considerably more structure than an arbitrary event-state algebra, it is important to note that they can still be described as special cases of event-state algebras. Thus we can apply our results for possibilities maps and invariants directly to these distributed algebras.

4.3.1 Distributed Event-State Algebras

Defn 4.3.1.1: Let $\mathcal{A} = (\mathcal{E}, \Sigma, \sigma, \tau)$ be an event-state algebra, let I be a finite *index set*, and let orig be a mapping orig: $\mathcal{E} \rightarrow$ I. We say that $\underline{\mathcal{A}}$ is distributed over I using orig provided that the following are true:

a. Σ is the Cartesian product of sets Σ_i, for i $\in$ I. We will use index i as the component name for set Σ_i.

b. σ is a vector of initial states, $\sigma_i \in \Sigma_i$, for i $\in$ I.

c. For each i $\in$ I, there is a local transition relation $\tau_i \subseteq \mathcal{E} \times \Sigma_i \times \Sigma_i$. τ_i must satisfy the following "local precondition" property: If e $\in \mathcal{E}$, s $\in \Sigma_i$, and orig(e) $\neq$ i, then $\tau_i(e)(s) \neq \perp$. Then τ is determined by the local transition relations as follows: $\tau = \{(e,s,t): (e,s.i,t.i) \in \tau_i, \forall i \in I\}$.

If orig(e) = i, then we say that component i is the originator of event e.

Because the transition relation of a distributed event-state algebra is defined by combining local transition relations for each component, the effect of each event on a component depends only on the current state of that component. It is possible for an event to affect several components, however. (Thus we are permitting an arbitrary "interconnection" of components through events.)

Although an event can have effects at several components, its precondition must be local to its originating component. Only the originator can control when one of its own events can occur.

In [Lynch82], a "local mapping" technique is explored for constructing a possibilities map from a distributed event-state algebra to another event-state algebra; the possibilities map is defined as the intersection of local possibilities maps from the states of all components in the distributed algebra.

4.3.2 Message-based Distributed Algebras

We now restrict distributed event-state algebras further to model the particular distributed environment of this thesis. The basic system components are *nodes*, with local state spaces and local event sets. All communication between nodes must flow through a distinguished system component, the *message buffer*. We define distinguished **send** and **receive** events for communications through the message buffer.

We give the message buffer a specific semantics: We postulate that messages are delivered in arbitrary order after they are sent, and that they can arrive any number of times (including 0) after they are sent. These assumptions allow us to model the message buffer as a *set* of messages (the set of all messages ever sent). It is never necessary to remove a message from this set, because we assume that messages can be duplicated and delayed arbitrarily.

Defn 4.3.2.1: Let $\mathcal{A} = (\mathcal{E}, \Sigma, \sigma, \tau)$ be an event-state algebra distributed over I using orig. Let Nodes be a finite set of *nodes*, let $\mathrm{Msgs}_{i,j}$ be a set of *messages* from node i to node j (i,j $\in$ Nodes), and let $\mathrm{Msgs} = \bigcup_{i,j} \mathrm{Msgs}_{i,j}$. We say that $\mathcal{A}$ is a message-based algebra over Nodes using Msgs if the following are true:

a. I = Nodes $\cup$ {buf}, where "buf" names the *message buffer* component.

b. $\Sigma_{\mathrm{buf}} = \mathcal{P}(\mathrm{Msgs})$ (i.e. the message buffer is a *set* of messages). Let BUF = Σ_{buf}

c. $\sigma.\mathrm{buf} = \emptyset$ (the message buffer is initially empty; thus no message can be received before it is sent).

d. Let Comm = {**send** M: M $\in$ Msgs} $\cup$ {**receive** M: M $\in$ Msgs} be the set of *communications events*. Then Comm $\subseteq \mathcal{E}$. If M $\in \mathrm{Msgs}_{i,j}$, then orig(**send** M) = i, orig(**receive** M) = buf: The originator of a **send** event is the source node for the message, and the originator of a **receive** event is the message buffer. (We regard the destination node for a message as passive in the communications process.)

e. If $e \in \mathcal{E}$ - Comm, and $i \neq$ orig(e), then $\tau_i(e)$ is the identity on Σ_i. Thus all "local" events (events not in Comm) must have only local effects. (Note that *preconditions* must be local by the definition of a distributed algebra).

f. If M $\in$ $\text{Msgs}_{i,j}$, then τ_k(**send** M) is the identity on Σ_k, for $k \neq$ buf,i. τ_i(**send** M) $\subseteq$ {(a,a): a $\in \Sigma_i$}. Thus although the sender of a message imposes a precondition on the sending of a message, the send has no effect on the sender's state.

g. τ_{buf}(**send** M) = {(b, b $\cup$ {M}): b $\in$ BUF}. Thus the effect of a **send** event on the buffer is to add the message to the buffer.

h. τ_k(**receive** M) is the identity on Σ_k, for $k \neq$ j,buf. τ_j(**receive** M)(a) $\neq \perp$, $\forall$a $\in$ Σ_j. Thus receipt of a message affects the state of the receiver, but the receiver cannot impose a precondition on receive events. (The originator of a **receive** event is the message buffer.)

i. τ_{buf}(**receive** M) = {(b,b): b $\in$ BUF $\wedge$ M $\in$ b}. Thus a receive event for a message can occur whenever that message is in the buffer. A receive event has no effect on the state of the message buffer, however. (Messages are never removed).

We stress that the message semantics we have chosen is not *inherent* in the distributed algebra framework; this semantics is simply convenient for describing our system. Our message-based model could be changed easily to provide for different communications semantics. For example, we could model a "reliable" communications system by making the message buffer an ordered list of messages, which only delivers messages from the head of the list and removes them upon delivery.

5. Proof Strategy

In the following chapters we will specify several levels of an event-state algebra hierarchy; these algebras model a distributed transaction system. The algebras are presented in top-down order: the top-level algebra (Level 0) is the "most abstract," and the bottom-level algebra (Level 7) is the "most concrete." At each level we specify the state, the initial state, the events, and the transition relation. At each level (except Level 0) we also specify a mapping from the new level to the previous (higher) level, and we show that this mapping is a possibilities map.

Our goal is to show that the orphan detection strategy which we outlined in Chapter 1 guarantees view-serializability. Thus our top-level model specifies our "correctness condition:" The Level 0 state is just the set of all action trees, and we define simple events to create, commit, and abort actions, and to perform an access. The only preconditions at Level 0 require that each state generated by any event be view-serializable.

At Level 1 we add a data ordering to the state (thus states are now *augmented* action trees). We impose preconditions on events to restrict the reachable states to view-serializable AAT's. We define the set of aborts "depended on" by an action; as one of our preconditions we require that each state generated by any event satisfy condition ANC-ABORT -- no action can depend on an abort of one of its ancestors. We then show that all reachable AAT's in Level 1 are view-serializable. Thus the obvious mapping from Level 1 to Level 0 is a possibilities map.

At Level 2 we remove the ANC-ABORT condition by adding a precondition to perform events. This precondition essentially states that an access should not see an abort dependency on an ancestor at the time it is performed. We show that the reachable states in Level 2 satisfy ANC-ABORT (using this new precondition); thus the obvious mapping from Level 2 to Level 1 is a possibilities map. We refer to this precondition as the "orphan detection" precondition.

Levels 0 - 2 are *global* state algebras, in that we regard the transaction system as operating on a single global state. These levels can be thought of as "centralized" interpretations of the events in a distributed action system. Lower levels progressively "distribute" this global state and localize the preconditions and effects of events.

At Level 3 we introduce "locations," which can be thought of as abstract nodes. Each action and each object has its own location. The information at a location consists of a (local) unlabeled action summary, plus the datastep ordering from the AAT. We define very simple "communications" events to transfer information to any location. We show that it is relatively simple to localize all preconditions *except for the orphan detection precondition.* The orphan detection condition must still be expressed in terms of the global AAT. The implication of this result is that our communications steps at Level 3 do not include enough information to completely localize orphan detection.

At Level 4 we introduce *value maps* -- a data structure which models the locks and versions of atomic objects. The Level 4 state consists of an AAT, a "local state" mapping from locations to UAS's, and a value map for each object. We regard the value map as a local data structure (conceptually each object has its own value map.) We replace some of the preconditions on perform events with preconditions on value maps, and we modify the transition effects of actions to update value maps appropriately.

At Level 5 we succeed in localizing the orphan detection precondition by piggybacking abort information on the create and commit communications events. This abort information models the DONE lists of our simplified orphan detection algorithm. The key invariant proved for Level 5 states that each location always has "enough" abort information.

Because all preconditions are localized at Level 5, the global AAT can be regarded as a "virtual" component of state. We project out this global state at Level 6, and we construct a trivial augmentation map between Level 6 and Level 5. Although the resulting algebra is "localized," it does not quite fit our definition of a "distributed" event-state algebra. To define a distributed event-state algebra, we must assign "locations" (abstract nodes) to physical nodes. An additional complication results from the simplicity of our communications events at Levels 3 - 6: The transfer of information caused by these events is considered *instantaneous* at these levels. For a distributed event-state algebra we must model arbitrary communications delays.

Level 7 presents a distributed event-state algebra. Many actions and objects can reside at a single node, and messages are sent asynchronously via a message buffer. In mapping from Level 7 to Level 6, we account for the communications delays in the message buffer by considering messages themselves to be abstract "locations." (One way to think of this device is to imagine that at Level 6 we can consider all communication events to be instantaneous, but all messages are sent via a third party. At Level 7, we "know" that this third party is really the message buffer, but at Level 6 this detail is not necessary.)

6. Global State Models

This chapter presents Levels 0 - 2 of the event-state algebra hierarchy. Level 0 describes our correctness condition: every action tree generated by the system must be view-serializable. Level 1 is a global state model based on AAT's. Level 1 develops the crucial link between view-serializability and orphan detection: We define the "aborts dependency set" for an action in an AAT, and we require at Level 1 that no action can depend on an abort of one of its ancestors. Informally this condition, which we call ANC-ABORT, means that no action can "know" that it is an orphan. At Level 1 we show that the ANC-ABORT condition (along with other preconditions on events) implies view-serializability.

The ANC-ABORT condition is imposed at Level 1 by requiring that the next state generated by each event satisfy ANC-ABORT. At Level 2 we replace this restriction with a single precondition on data accesses, and we show that this precondition suffices to guarantee ANC-ABORT.

We also make use of an auxiliary algebra, which we call "Level A" (denoted La). Level A consists of Level 1 without the ANC-ABORT restriction. Thus Levels 1 and 2 are both logically "below" Level A. The advantage of using this auxiliary level is that we can easily construct a possibilities map from Level 2 to Level A; we will then use Level A invariants in showing that there is a possibilities map from Level 2 to Level 1.

We will use the following distinguished symbols to define the initial states of the algebras:

T_0 denotes the trivial AAT containing only vertex U with status 'active', and an empty data ordering:

$\text{vertices}_{T_0} = \{U\}$
$\text{status}_{T_0}(U) = \text{'active'}$
$\text{label}_{T_0} = \emptyset$
$\text{data}_{T_0} = \emptyset$

T_i = erase(T_0) (an action tree), and T_u = unlabel(T_i) (a UAS).

6.1 Level 0 Algebra

The Level 0 state consists of a (global) action tree. The events at Level 0 are just those needed to create an action tree: we define events to create an action, commit and abort an action, and perform an access with a given value (this value gives the label of the datastep in the action tree). The only constraint on validity of an execution sequence at Level 0 is that the resulting action tree must be view-serializable.

$$L0 = (\mathcal{E}_0, \Sigma_0, \sigma_0, \tau_0)$$

$\mathcal{E}_0$ = {create A, commit A, abort A, perform A,u} (see below).

Σ_0 is the set of all action trees.

$\sigma_0 = T_i$, the trivial action tree.

τ_0, the transition relation, is specified below via preconditions and transition effects for each event:

Let the current state be T. For each event, we give the transition function which maps T → T1. The precondition for each event is a logically a condition on T, the current state, but we specify it as a condition on T1. (Since T uniquely determines T1, a condition on T1 maps directly into a condition on T.) The single precondition for each event requires that the next state (T1) be view-serializable. Let VSR denote the set {T: T is a view-serializable action tree}.

1. <u>create A</u> $(A \in act - \{U\})$

PRECONDITIONS:

a. $T1 \in VSR$

TRANSITIONS:

a. $vertices_{T1} \leftarrow vertices_T \cup \{A\}$

b. $status_{T1}(A) \leftarrow$ 'active'

2. <u>commit A</u> $(A \in act - \{U\} - accesses)$

PRECONDITIONS:

a. T1 ∈ VSR

TRANSITIONS:

a. $\text{status}_{T1}(A) \leftarrow$ 'committed'

3. abort A $\quad (A \in \text{act} - \{U\})$

PRECONDITIONS:

a. T1 ∈ VSR

TRANSITIONS:

a. $\text{status}_{T1}(A) \leftarrow$ 'aborted'

4. perform A,u $\quad (A \in \text{accesses}(x),\ u \in \text{values}(x))$

PRECONDITIONS:

a. T1 ∈ VSR

TRANSITIONS:

a. $\text{status}_{T1}(A) \leftarrow$ 'committed'

b. $\text{label}_{T1}(A) \leftarrow u$

The following lemma justifies our statement that L0 defines our correctness condition, because all reachable states in L0 are view-serializable action trees.

Lemma 6.1.1: Let $T \in \mathcal{R}_0$. Then $T \in$ VSR.

Proof: Let $T = T_i v$, for some $v \in \mathcal{V}_0$. If $v \neq \Lambda$, then $T = T'e$ for some $e \in \mathcal{E}_0$, $T' \in \text{PRE}_0(e)$, and by the VSR precondition for e, $T \in$ VSR. If $v = \Lambda$, then $T = T_i$ which is trivially in VSR. ∎

6.2 Level 1 Algebra and Mapping h_{10}

The Level 1 state consists of a (global) AAT. The events are identical to those defined at Level 0, but we modify the preconditions as we begin to specify in detail how the transaction system functions.

$$L1 = (\mathcal{E}_1, \Sigma_1, \sigma_1, \tau_1)$$

$\mathcal{E}_1 = \mathcal{E}_0 =$ {create A, commit A, abort A, perform A,u}.

Σ_1 is the set of all augmented action trees.

$\sigma_1 = T_0$, the trivial AAT.

τ_1, the transition relation, is specified below via preconditions and transition effects for each event.

We will define a condition, ANC-ABORT, on AAT's, which essentially states that an action cannot know that it is an orphan. We include a precondition for each event in L1 which requires that the *next state* generated by this event must satisfy ANC-ABORT. It will follow trivially that ANC-ABORT is satisfied by all reachable states in L1.

6.2.1 Aborts Dependencies and Condition ANC-ABORT

We want to develop a condition which will rule out execution sequences in which orphans see "inconsistent" data. To devise a condition which can distinguish "bad" orphans from orphans which are not dangerous, we define the set of aborts upon which an action "depends." The ANC-ABORT condition simply states that an action cannot depend upon the abort of any of its ancestors.

Informally, an action depends on any abort which allowed the action to proceed. Because of sequential dependencies, any abort of a sequentially preceding sibling is depended on by its following siblings and their descendants. A parent also depends on the aborts of any of its children. Any abort which "releases a lock" on an object subsequently read by an action is depended upon by that action. Our Level 1 model does not have explicit "locks"; locks and versions are represented by the entire action tree. (Precondition P1.4b below is essentially a "lock" condition which says that two actions (at any level) cannot interfere on the same object: one must either commit or abort before the other is allowed to proceed. Precondition P1.4c is essentially a "current version" condition which says that the current version seen by a datastep is the result of all preceding accesses which are visible to it.)

An action also depends on all the aborts depended on by committed actions which might pass information to it (which for our purposes will be considered all the actions in its view set). Thus the aborts dependency set for an action is defined as the union over all actions in its view set of the "immediate aborts" preceding those actions.

Defn 6.2.1.1: Let T be an AAT, A $\in$ vertices_T. We define the <u>aborts dependency set of A in T</u> as follows:

$$\underline{\text{ABORTS}_T}(A) = \bigcup_{B \in \text{vset}_T(A)} \text{i-precedes}_T(B)$$

We define the set ANC-ABORT as the set of all AAT's in which no action depends upon the abort of an ancestor:

Defn 6.2.1.2: ANC-ABORT = $\{T: \forall A \in \text{vertices}_T, \text{anc}(A) \cap \text{ABORTS}_T(A) = \emptyset\}$.

We also define a "sequential aborts set" which represents all the aborts upon which an action depends *when it is first created.*

Defn 6.2.1.3: Let T be an AAT, A $\in$ vertices_T. We define the <u>sequential aborts dependency set of A in T</u> as follows:

$$\underline{\text{SEQ-ABORTS}_T}(A) = \text{i-anc-seq}_T(A) \cup \bigcup_{B \in \text{v-anc-seq}_T(A)} \text{ABORTS}_T(B)$$

The following lemma relates the sequential aborts set of an action to the sequential aborts set of its parent:

Lemma 6.2.1.4: Let T be an AAT, and A $\in$ vertices_T. If A $\neq$ U, then

$$\text{SEQ-ABORTS}_T(A) = \text{SEQ-ABORTS}_T(\text{parent}(A)) \cup \bigcup_{B \in \text{v-seq}_T(A)} \text{ABORTS}_T(B) \quad \cup \ \text{i-seq}_T(A).$$

And $\text{SEQ-ABORTS}_T(U) = \emptyset$.

Proof: It is obvious that $\text{SEQ-ABORTS}_T(U) = \emptyset$. Take A $\neq$ U. By definition of SEQ-ABORTS,

$$\text{SEQ-ABORTS}_T(A) = \text{i-anc-seq}_T(A) \cup \bigcup_{B \in \text{v-anc-seq}_T(A)} \text{ABORTS}_T(B) .$$

But i-anc-seq$_T$(A) = i-seq$_T$(A) ∪ i-anc-seq$_T$(parent(A)), and v-anc-seq$_T$(A) = v-seq$_T$(A) ∪ v-anc-seq$_T$(parent(A)). Thus

$$\text{SEQ-ABORTS}_T(A) = \text{i-seq}_T(A) \cup \text{i-anc-seq}_T(\text{parent}(A)) \cup \bigcup_{B \in \text{v-seq}_T(A)} \text{ABORTS}_T(B) \cup \bigcup_{B \in \text{v-anc-seq}_T(\text{parent}(A))} \text{ABORTS}_T(B) .$$

$$= \text{SEQ-ABORTS}_T(\text{parent}(A)) \cup \bigcup_{B \in \text{v-seq}_T(A)} \text{ABORTS}_T(B) \cup \text{i-seq}_T(A). \blacksquare$$

The following lemma relates the flow of information via view sets to the flow of abort information via ABORTS sets:

Lemma 6.2.1.5: Let T be an AAT, A ∈ vertices$_T$, B ∈ vset$_T$(A), then

$$\text{ABORTS}_T(B) \subseteq \text{ABORTS}_T(A)$$

Proof: $\text{ABORTS}_T(A) = \bigcup_{C \in \text{vset}_T(A)} \text{i-precedes}_T(C)$, while $\text{ABORTS}_T(B) = \bigcup_{C \in \text{vset}_T(B)} \text{i-precedes}_T(C)$.

But if B ∈ vset$_T$(A), then vset$_T$(B) ⊆ vset$_T$(A) by Lemma 2.3.3a. The lemma follows directly.

∎

The definition of view sets as the closure under v-precedes$_T$ allows us to write a recursive expression for ABORTS$_T$:

Lemma 6.2.1.6: Let T be an AAT, A ∈ vertices$_T$, then

$$\text{ABORTS}_T(A) = \text{i-precedes}_T(A) \cup \bigcup_{B \in \text{v-precedes}_T(A)} \text{ABORTS}_T(B)$$

Proof: $\text{vset}_T(A) = \{A\} \cup \text{v-precedes}^+_T(A)$

$$= \{A\} \cup \bigcup_{B \in \text{v-precedes}_T(A)} \text{vset}_T(B)$$

The Lemma follows directly. ∎

Since action trees are always finite, we can use this recursive form in inductive proofs of properties of aborts sets if we show that tracing back the v-precedes$_T$ relation will not result in cycles, i.e. that ∀A ∈ vertices$_T$, A ∉ v-precedes^+_T(A). (If v-precedes$_T$ were acyclic, then the induction might not be well-founded.) We will prove below that v-precedes$_T$ is acyclic for all reachable trees in La (and hence for all reachable trees in L1).

6.2.2 Specification of Event Preconditions and Transitions for L1

Let the current state be T. For each event, we give the the transition function which maps T $\rightarrow$ T1. Preconditions are specified as a function of T, except for the ANC-ABORT condition which requires that the *next state* be in ANC-ABORT.

1. create A (A $\in$ act - {U})

PRECONDITIONS:

a. A $\notin$ vertices_T

b. parent(A) $\in$ active_T

c. (B,A) $\in$ seq, B $\neq$ A $\Rightarrow$ B $\in$ done_T

d. T1 $\in$ ANC-ABORT

TRANSITIONS:

a. $\text{vertices}_{T1} \leftarrow \text{vertices}_T \cup \{A\}$

b. $\text{status}_{T1}(A) \leftarrow$ 'active'

2. commit A (A $\in$ act - {U} - accesses)

PRECONDITIONS:

a. A $\in$ active_T

b. $\text{children}_T(A) \subseteq \text{done}_T$

c. T1 $\in$ ANC-ABORT

TRANSITIONS:

a. $\text{status}_{T1}(A) \leftarrow$ 'committed'

3. abort A $(A \in act - \{U\})$

PRECONDITIONS:

a. $A \in active_T$

b. $T1 \in ANC\text{-}ABORT$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'aborted'

4. perform A.u $(A \in accesses(x),\ u \in values(x))$

PRECONDITIONS:

a. $A \in active_T$

b. $B \in datasteps_T(x) \Rightarrow B \in visible_T(A,x) \vee B \in dead_T(A,x)$

c. $u = result(x,s)$, where $s = \langle\langle visible_T(A,x); data_T\rangle\rangle$

d. $T1 \in ANC\text{-}ABORT$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'committed'

b. $label_{T1}(A) \leftarrow u$

c. $data_{T1} \leftarrow data_T \cup \{(B,A): B \in datasteps_T(x)\} \cup \{(A,A)\}$

6.2.3 Specification of Mapping h_{10}

We define the mapping h_{10}: L1 → L0 in the obvious way. Our goal, of course, is to show that this mapping is a possibilities map.

State Mapping

$h_{10}: \Sigma_1 \rightarrow \Sigma_0$ is defined by $h_{10}(T) = erase(T)$, $\forall T \in \Sigma_1$.

Event Mapping

h_{10}: $\mathcal{E}_1 \rightarrow \mathcal{E}_0^*$ is the identity map on events.

6.2.4 Proof Strategy for Showing h_{10} is a Possibilities Map

We can show easily that h_{10} preserves initial states and transitions:

Lemma 6.2.4.1: h_{10} preserves initial states.

Proof: $h_{10}(T_0) = \mathrm{erase}(T_0) = T_i$, by definition. ▮

Lemma 6.2.4.2: h_{10} preserves transitions.

Proof: It is obvious by inspection that h_{10} preserves transitions, since transitions for all events are identical at levels L0 and L1 (except for transition T1.4c, which involves the data ordering -- but data_T is projected out by the state mapping). ▮

Showing that h_{10} preserves preconditions is more difficult. We use the following lemma to reduce this problem to a view-serializability condition on reachable states in L1:

Lemma 6.2.4.3: Suppose that for all $T \in \mathcal{R}_1$, T is view-serializable. Then h_{10} preserves preconditions.

Proof: To show that h_{10} preserves preconditions, we must show that

$T \in PRE_1(e) \cap \mathcal{R}_1, h_{10}(T) \in \mathcal{R}_0 \;\Rightarrow\; h_{10}(T) \in PRE_0(h_{10}(e))$.

But the only precondition at Level 0 is that the *next state* must be in VSR. Thus

$h_{10}(T) \in PRE_0(h_{10}(e)) \;\Leftrightarrow\; h_{10}(T)h_{10}(e) \in VSR$.

Since h preserves transitions, $h_{10}(T)h_{10}(e) = h_{10}(Te) = \mathrm{erase}(Te)$. Thus we must show that $\mathrm{erase}(Te) \in VSR$. Since view-serializability of AAT's is defined to be view-serializability of the corresponding action tree, we must show that

$T \in PRE_1(e) \cap \mathcal{R}_1, \mathrm{erase}(T) \in \mathcal{R}_0 \;\Rightarrow\;$ Te is view-serializable.

But $T \in PRE_1(e) \cap \mathcal{R}_1 \;\Rightarrow\; Te \in \mathcal{R}_1$. Thus it suffices to show that all reachable states in L1 are view-serializable. ▮

View-serializability of reachable states is thus our main theorem for L1, which will imply that h_{10} preserves preconditions (and is thus a possibilities map). We state this theorem here, although its proof will be given in several stages:

Theorem 6.2.4.4: Let $T \in \mathcal{T}_1$. Then T is view-serializable.

The proof of this theorem consists of showing that for each action $A \in \text{vertices}_T$, $\text{vtree}_T(A)$ is a serializable view tree for A. The proof that $S = \text{vtree}_T(A)$ is a serializable view tree is given in three subordinate lemmas which show that (1) S is a view tree for A in T, (2) S is version-compatible, and (3) there are no cycles (of length 2 or greater) in $\text{seq}_S \cup \text{sibling-data}_S$. By Theorem 2.5.1, it follows that S is a serializable view tree for A. We state these lemmas here, although the proofs are deferred to later sections.

Lemma 6.2.4.5: Let $T \in \mathcal{T}_1$, let $A \in \text{vertices}_T$, and let $S = \text{vtree}_T(A)$. Then S is a view tree for A in T.

Lemma 6.2.4.6: Let $T \in \mathcal{T}_1$, let $A \in \text{vertices}_T$, and let $S = \text{vtree}_T(A)$. Then S is version-compatible.

Lemma 6.2.4.7: Let $T \in \mathcal{T}_1$, let $A \in \text{vertices}_T$, and let $S = \text{vtree}_T(A)$. Then $\text{seq}_S \cup \text{sibling-data}_S$ has no cycles of length two or greater.

6.3 Auxiliary Algebra La

We define an "auxiliary" event-state algebra, La. (La is "auxiliary" because it is not part of our main event-state algebra hierarchy.) La is identical to L1, except that the ANC-ABORT preconditions on events (preconditions P1.1d, P1.2c, P1.3b, and P1.4d) are omitted.

We define the trivial mapping h_{1a}: L1 → La as the identity map on states and events.

Theorem 6.3.1: h_{1a} is a possibilities map.

Proof: Since initial states are identical in L1 and La, and h_{1a} is the identity on states, h_{1a}

preserves initial states. Since all transitions and preconditions in Level 1 also appear at Level A, h_{1a} must preserve transitions and preconditions. Thus h_{1a} is a possibilities map, by Lemma 4.2.2.6. ∎

Since this mapping fixes T (it must fix T since T is the entire state), we will show that all invariants (and pair-invariants) for La are invariant (or pair-invariant) for L1.

We prove below several basic lemmas for algebra La. We will then apply these results to the proofs of Lemmas 6.2.4.5, 6.2.4.6, and 6.2.4.7.

The advantage of defining La is that we will also construct a trivial possibilities map between algebra L2 and algebra La. We will thus be able to apply Level A invariants directly to Level 2, and we will use these invariants to show that h_{21} (defined below) is a possibilities map.

6.3.1 Basic Lemmas for La

6.3.1.1 Invariants and Pair-Invariants for La

Lemma 6.3.1.1.1: Let $(T,T1) \in \mathcal{T}_a^{(2)}$, and let $A \in \text{vertices}_T$. Then the following are true:

a. $\text{vertices}_T \subseteq \text{vertices}_{T1}$, $\text{committed}_T \subseteq \text{committed}_{T1}$, $\text{aborted}_T \subseteq \text{aborted}_{T1}$, $\text{data}_T \subseteq \text{data}_{T1}$

b. If $A \in \text{datasteps}_T$, then $\text{label}_T(A) = \text{label}_{T1}(A)$

c. If $A \in \text{datasteps}_T$ and $(B,A) \in \text{data}_{T1}$, then $(B,A) \in \text{data}_T$

d. $\text{visible}_T(A) \subseteq \text{visible}_{T1}(A)$

e. $\text{dead}_T(A) \subseteq \text{dead}_{T1}(A)$

f. If A is live in T1, then A is live in T

g. If A is dead in T, then A is dead in T1 and $\{\text{crucial}_{T1}(A)\} \leq \{\text{crucial}_T(A)\}$

h. $\text{v-anc-seq}_{T1}(A) = \text{v-anc-seq}_T(A)$, $\text{i-anc-seq}_{T1}(A) = \text{i-anc-seq}_T(A)$

Proof: Straightforward. ∎

Lemma 6.3.1.1.2: Let $T \in \mathcal{T}_a$. Then the following invariants hold:

a. T is an AAT, i.e. $A \in \text{vertices}_T \Rightarrow \text{parent}(A) \in \text{vertices}_T$

b. If $A \in \text{vertices}_T$ and $(B,A) \in \text{seq}$ and $B \neq A$, then $B \in \text{done}_T$

c. If $A \in \text{vertices}_T$ and $\text{parent}(A) \in \text{committed}_T$, then $A \in \text{done}_T$

d. $U \in \text{active}_T$

e. If $(B,A) \in \text{data}_T$, then $B \in \text{visible}_T(A) \ \vee \ B \in \text{dead}_T(A)$

f. If $A \in \text{committed}_T$ and $B \in \text{desc}(A) \cap \text{vertices}_T$, then $B \in \text{visible}_T(A) \ \vee \ B \in \text{dead}_T(A)$

g. If $(B,A) \in \text{i-data}_T$, then $\text{crucial}_T(B)$ is defined, and $\text{crucial}_T(B) \in \text{desc}(A{\downarrow}B)$

Proof: All are obvious except for (e) and (f) ((g) follows directly from (e)):

e) If $B = A$ then the result is immediate. If $B \neq A$, then

Let $T = T_0 v$, where $v \in \mathcal{V}_1$ can be written as $\varphi \pi \psi$, with π = perform A,u.
Let $T1 = T_0 \varphi$, and let $T2 = T_0 \varphi \pi$.

By Lemma 6.3.1.1.1c, $(B,A) \in \text{data}_{T2} \Rightarrow B \in \text{datasteps}_{T1}(x)$. By precondition Pa.4b for perform, $B \in \text{visible}_{T1}(A.x) \vee B \in \text{dead}_{T1}(A,x)$.

$B \in \text{visible}_{T1}(A,x) \Rightarrow B \in \text{visible}_T(A,x)$ (by Lemma 6.3.1.1.1d), $\Rightarrow$ $B \in \text{visible}_T(A)$.

$B \in \text{dead}_{T1}(A,x) \Rightarrow B \in \text{dead}_T(A,x)$ (by Lemma 6.3.1.1.1e), $\Rightarrow$ $B \in \text{dead}_T(A)$.

f) If $B = A$, then the result is immediate. So assume $B \in \text{prop-desc}(A)$, and assume $B \neq \text{visible}_T(A)$. Let $C \in \text{prop-desc}(A) \cap \text{anc}(B)$ be the highest ancestor of B which is not committed. Then $\text{parent}(C) \in \text{committed}_T \Rightarrow C \in \text{done}_T$ (by Lemma 6.3.1.1.2c). But $C \notin \text{committed}_T$ by assumption $\Rightarrow C \in \text{aborted}_T$. ▮

Lemma 6.3.1.1.3: Let $(T,T1) \in \mathcal{R}_a^{(2)}$, and let $A \in \text{committed}_T$. Then the following are true:

a. $\text{children}_{T1}(A) = \text{children}_T(A)$

b. v-child$_{T1}$(A) = v-child$_T$(A), i-child$_{T1}$(A) = i-child$_T$(A)

c. v-data$_{T1}$(A) = v-data$_T$(A), i-data$_{T1}$(A) = i-data$_T$(A),
v-data-anc$_{T1}$(A) = v-data-anc$_T$(A)

d. i-data-anc$_{T1}$(A) $\leq$ i-data-anc$_T$(A)

e. v-precedes$_{T1}$(A) = v-precedes$_T$(A), vset$_{T1}$(A) = vset$_T$(A)

f. i-precedes$_{T1}$(A) $\leq$ i-precedes$_T$(A)

Proof:

a) Clearly children$_T$(A) $\subseteq$ children$_T$(A). Suppose B $\in$ children$_{T1}$(A) - children$_T$(A). (We can assume A $\notin$ accesses.)

Let T1 = T$_0$v, where v $\in \mathcal{T}_a$ can be written as $\varphi\pi\psi\rho\gamma$, with π = commit A, ρ = create B. Let T2 = T$_0\varphi\pi\psi$.

Then A $\in$ committed$_{T2}$. But precondition Pa.1b requires that A $\in$ active$_{T2}$, a contradiction.

b) Follows directly from (a) and Lemma 6.3.1.1.1d

c) Because any datastep which occurs after perform A,u must follow A in the data ordering, v-data$_{T1}$(A) $\cup$ i-data$_{T1}$(A) = v-data$_T$(A) $\cup$ i-data$_T$(A). But i-data$_T$(A) $\subseteq$ dead$_T$(A) by Lemma 6.3.1.1.2c, and dead$_T$(A) $\subseteq$ dead$_{T1}$(A) by Lemma 6.3.1.1.1e. Thus i-data$_T$(A) $\subseteq$ i-data$_{T1}$(A).

But v-data$_T$(A) $\subseteq$ v-data$_{T1}$(A) by Lemma 6.3.1.1.1d. It follow directly that v-data$_T$(A) = v-data$_{T1}$(A), and i-data$_T$(A) = i-data$_{T1}$(A). Equality of v-data directly implies equality of v-data-anc.

d) Follows directly from (c) and Lemma 6.3.1.1.1g

e) Equality of v-precedes$_{T1}$(A) and v-precedes$_T$(A) follows directly from parts (b) and (c) and from Lemma 6.3.1.1.1h. To show that vset$_{T1}$(A) = vset$_T$(A), we can argue inductively since B $\in$ v-precedes$_T$(A) $\Rightarrow$ B $\in$ committed$_T$ by Lemma 2.3.2.

f) Follows directly from parts (b) and (d) and from Lemma 6.3.1.1.1h. ∎

Lemma 6.3.1.1.4: Let $T \in \mathcal{T}_a$, $A \in \text{committed}_T$. Then $\text{i-precedes}_T(A) \subseteq \text{aborted}_T$.

Proof: Let $B \in \text{i-precedes}_T(A)$.

$B \in \text{i-anc-seq}_T \Rightarrow B \in \text{done}_T$ by Lemma 6.3.1.1.2b, $\Rightarrow B \in \text{aborted}_T$ (since $B \notin \text{visible}_T(A)$).

$B \in \text{i-child}_T \Rightarrow B \in \text{done}_T$ by Lemma 6.3.1.1.2c, $\Rightarrow B \in \text{aborted}_T$ (since $B \notin \text{visible}_T(A)$).

$B \in \text{i-data-anc}_T \Rightarrow B = \text{crucial}_T(b)$, for $b \in \text{i-data}_T(A)$. By Lemma 6.3.1.1.2g, B is defined, $\Rightarrow B \in \text{aborted}_T$. ▮

6.3.1.2 Event Orderings in La

This section presents some constraints on the ordering of events in valid execution sequences for La. In the following lemmas (and in the proofs that follow) we will simplify our notation by referring to both "perform A,u" events and "commit A" events as "commit A." This convention causes no complications; it requires only that we realize that events written as "commit A" might refer to datasteps.

Lemma 6.3.1.2.1: Let $v \in \mathcal{V}_a$ be a valid execution sequence from La, then $\underset{v}{\rightarrow}$ is acyclic -- i.e., no event can be repeated in a valid execution sequence.

Proof: Suppose event e could be repeated in a valid execution, v, i.e. $v = a \cdot e \cdot b \cdot c \cdot c \in \mathcal{V}_a$. Let $T1 = T_0ae$, $T2 = T_0aeb$. By Lemma 6.3.1.1.1,

$e = \text{create } A \Rightarrow A \in \text{vertices}_{T2}$.

$e = \text{commit } A \Rightarrow A \in \text{committed}_{T2}$.

$e = \text{abort } A \Rightarrow A \in \text{aborted}_{T2}$.

But by the preconditions for events,

$e = \text{create } A$ requires $A \notin \text{vertices}_{T2}$ (Pa.1a).

$e = \text{commit } A$ requires $A \in \text{active}_{T2}$ (Pa.2a and Pa.4a).

$e = \text{abort } A$ requires $A \in \text{active}_{T2}$ (Pa.3a).

Thus no event can be repeated. ▮

Lemma 6.3.1.2.2: Let $v \in \mathcal{T}_a$, $T = T_0v$, $A \in \text{committed}_T$, then

$$\text{create } A \xrightarrow[v]{} \text{commit } A$$

Proof: v can be written as $\varphi\pi\psi$, with $\pi = \text{commit } A$.

Let $T1 = T_0\varphi$.

Precondition Pa.2a (or Pa.4a if $A \in$ accesses) requires $A \in \text{active}_{T1}$,

$\Rightarrow$ create $A \in \varphi$,

$\Rightarrow$ create $A \xrightarrow[v]{}$ commit A. ∎

Lemma 6.3.1.2.3: Let $v \in \mathcal{T}_a$, $T = T_0v$, $A \in \text{aborted}_T$, then

$$\text{create } A \xrightarrow[v]{} \text{abort } A$$

Proof: Similar to the proof of 6.3.1.2.2 above. ∎

Lemma 6.3.1.2.4: Let $v \in \mathcal{T}_a$, $T = T_0v$, $(B,A) \in \text{data}_T$, $B \neq A$, then

$$\text{commit } B \xrightarrow[v]{} \text{commit } A$$

Proof: v can be written as $\varphi\pi\psi$, with $\pi = \text{commit } A$ (= perform A,u).

Let $T1 = T_0\varphi$, and let $T2 = T_0\varphi\pi$.

By Lemma 6.3.1.1.1c, $(B,A) \in \text{data}_{T2}$,

$\Rightarrow B \in \text{committed}_{T1}$,

$\Rightarrow$ commit $B \xrightarrow[v]{} \pi$ (= commit A). ∎

Lemma 6.3.1.2.5: Let $v \in \mathcal{T}_a$, $T = T_0v$, $A \in \text{datasteps}_T$, $B \in \text{v-data}_T(A)$, then

$$\text{commit } A{\downarrow}B \xrightarrow[v]{} \text{commit } A$$

Proof: $A \in \text{datasteps}_T \Rightarrow$ commit $A \in v$.

Thus v can be written as $\varphi\pi\psi$, with $\pi = \text{commit } A$ (= perform A,u).

Let $T1 = T_0\varphi$, and let $T2 = T_0\varphi\pi$.

By Lemma 6.3.1.1.1c, $(B,A) \in \text{data}_{T2}$.

By Lemma 6.3.1.1.2e, $B \in \text{visible}_{T2}(A) \ \vee \ B \in \text{dead}_{T2}(A)$.

If $B \in \text{dead}_{T2}(A)$ then $B \in \text{dead}_{T}(A)$, $\Rightarrow$ $B \notin \text{visible}_{T}(A)$, a contradiction.

Thus $B \in \text{visible}_{T2}(A)$,

$\Rightarrow$ $A{\downarrow}B \in \text{committed}_{T2}$,

$\Rightarrow$ commit $A{\downarrow}B \in \varphi$,

$\Rightarrow$ commit $A{\downarrow}B \underset{v}{\rightarrow}$ commit A. ∎

Lemma 6.3.1.2.6: Let $v \in \mathcal{T}_a$, $T = T_0 v$, $A \in \text{vertices}_T$, $A' \in \text{prop-anc}(A) - \{U\}$, then

create A' $\underset{v}{\rightarrow}$ create A

Proof: $A \in \text{vertices}_T \Rightarrow$ create $A \in v$.

Thus v can be written as $\varphi\pi\psi$, with π = create A.

Let $T1 = T_0\varphi$, and let $T2 = T_0\varphi\pi$.

By precondition Pa.1b, $\text{parent}(A) \in \text{active}_{T1}$,

$\Rightarrow$ create parent(A) $\underset{v}{\rightarrow}$ create A (unless parent(A) = {U}).

The Lemma follows by an obvious induction. ∎

Lemma 6.3.1.2.7: Let $v \in \mathcal{T}_a$, $T = T_0 v$, $A \in \text{vertices}_T$, $B \in \text{v-anc-seq}_T(A)$, then

commit B $\underset{v}{\rightarrow}$ create A

Proof: $A \in \text{vertices}_T \Rightarrow$ create $A \in v$.

$B \in \text{v-anc-seq}_T(A) \Rightarrow \exists A' \in \text{anc}(A)$: $(B,A') \in \text{seq}$.

By Lemma 6.3.1.2.6, create A' $\underset{v}{\rightarrow}{}^{*}$ create A.

v can be written as $\varphi\pi\psi$, with π = create A'.

Let $T1 = T_0\varphi$, and let $T2 = T_0\varphi\pi$.

By precondition Pa.1c, $B \in done_{T1}$,

$\Rightarrow$ commit A $\underset{v}{\rightarrow}$ create A' $\underset{v}{\rightarrow}{}^{*}$ create A. ▮

Lemma 6.3.1.2.8: Let $v \in \mathcal{T}_{a}$, $T = T_0 v$, $A \in vertices_T$, $B \in$ i-anc-seq$_T$(A), then

abort B $\underset{v}{\rightarrow}$ create A

Proof: Similar to the proof of Lemma 6.3.1.2.7 above. ▮

Lemma 6.3.1.2.9: Let $v \in \mathcal{T}_{a}$, $T = T_0 v$, $A \in committed_T$, $B \in$ v-prop-desc$_T$(A), then

commit B $\underset{v}{\rightarrow}$ commit A

Proof: $A \in committed_T \Rightarrow$ commit $A \in v$. Note that since A has a proper descendant, $A \notin$ accesses. Assume that $B \in$ v-child$_T$(A); the Lemma follows from this case by an obvious induction.

v can be written as $\varphi\pi\psi$, with π = commit A.
Let $T1 = T_0\varphi$, and let $T2 = T_0\varphi\pi$.

But B cannot be created after A has committed, so $B \in vertices_{T1}$.

By precondition Pa.2b, $B \in done_{T1}$,

$\Rightarrow$ commit $B \in \varphi$,

$\Rightarrow$ commit B $\underset{v}{\rightarrow}$ commit A. ▮

Lemma 6.3.1.2.10: Let $v \in \mathcal{T}_{a}$, $T = T_0 v$, $A \in committed_T$, $B \in$ i-child$_T$(A), then

abort B $\underset{v}{\rightarrow}$ commit A

Proof: Similar to the proof of Lemma 6.3.1.2.9 above. ▮

Lemma 6.3.1.2.11: Let $v \in \mathcal{T}_{a}$, $T = T_0 v$, $A,B \in committed_T$, $B \in$ v-precedes$^{+}_{T}$(A), then

commit B $\underset{v}{\rightarrow}$ commit A

Proof: We show $C \in$ v-precedes$_T(B) \Rightarrow$ commit $C \underset{v}{\rightarrow}$ commit B. The Lemma follows by an obvious induction.

$C \in$ v-precedes$_T(B) \Rightarrow$

$C \in$ v-anc-seq$_T(B) \lor C \in$ v-child$_T(B) \lor C \in$ v-data-anc$_T(B)$.

If $C \in$ v-anc-seq$_T(B)$, then

commit $C \underset{v}{\rightarrow}$ create B by Lemma 6.3.1.2.7.

But create $B \underset{v}{\rightarrow}$ commit B by Lemma 6.3.1.2.2,

$\Rightarrow$ commit $C \underset{v}{\rightarrow}$ commit B.

If $C \in$ v-child$_T(B)$, then

commit $C \underset{v}{\rightarrow}$ commit B by Lemma 6.3.1.2.9.

If $C \in$ v-data-anc$_T(B)$, then

$C = B{\downarrow}c$, where $c \in$ v-data$_T(B)$,

$\Rightarrow$ commit $C \underset{v}{\rightarrow}$ commit B by Lemma 6.3.1.2.5. ∎

Lemma 6.3.1.2.12: Let $v \in \Upsilon_a$, $T = T_0 v$, $A, B \in$ committed$_T$, $B \in$ vset$_T^+(A)$, then

commit $B \underset{v}{\rightarrow}{}^*$ commit A

Proof: Immediate corollary of Lemma 6.3.1.2.11. ∎

Lemma 6.3.1.2.13: Let $v \in \Upsilon_a$, $T = T_0 v$, $A \in$ committed$_T$, $B \in$ i-precedes$_T(A)$, then

create $B \underset{v}{\rightarrow}$ commit A

Proof: $B \in$ i-precedes$_T(A) \Rightarrow$

$B \in$ i-anc-seq$_T(A) \lor B \in$ i-child$_T(A) \lor B \in$ i-data-anc$_T(A)$.

If $B \in$ i-anc-seq$_T(A)$, then abort $B \underset{v}{\rightarrow}$ create A by Lemma 6.3.1.2.8, and

create $B \underset{v}{\rightarrow}$ abort B, create $A \underset{v}{\rightarrow}$ commit A,

$\Rightarrow$ create $B \underset{v}{\rightarrow}$ commit A.

If $B \in$ i-child$_T(A)$, then abort $B \underset{v}{\rightarrow}$ commit A by Lemma 6.3.1.2.10, and create $B \underset{v}{\rightarrow}$ abort B,

$\Rightarrow$ create $B \underset{v}{\rightarrow}$ commit A.

If $B \in$ i-data-anc$_T(A)$, then $B =$ crucial$_T(B')$ for some $(B',A) \in$ data$_T$. Thus $B \in$ desc(B').

But by Lemma 6.3.1.2.4, commit $B' \underset{v}{\rightarrow}$ commit A, and by Lemma 6.3.1.2.6, create $B \underset{v}{\rightarrow}$ create $B' \underset{v}{\rightarrow}$ commit B'

$\Rightarrow$ create $B \underset{v}{\rightarrow}$ commit A. ∎

Lemma 6.3.1.2.14: Let $v \in \mathcal{T}_a$, $T = T_0v$, $A,B \in$ committed$_T$, $B \in$ vset$_T(A)$, $C \in$ i-precedes$_T(B)$, then

create $B \underset{v}{\rightarrow}$ commit A

Proof: Immediate corollary of Lemmas 6.3.1.2.12 and 6.3.1.2.13. ∎

6.3.2 Version-Compatibility in La

Lemma 6.2.4.6 states that if T is a reachable AAT in L1, and $A \in$ vertices$_T$, then vtree$_T(A)$ is version-compatible. In this section we develop two lemmas which will be used in the proof of Lemma 6.2.4.6. First we show that if T is any AAT which is version-compatible, then any restriction of T to a v-data$_T$-closed set is also version-compatible. We then show that any reachable tree in La is version-compatible. We will show in a later section that for any reachable tree, T, in L1, vtree$_T(A)$ is a (backed up) restriction of T to a v-data$_T$-closed set, which will complete the proof of Lemma 6.2.4.6.

Lemma 6.3.2.1: Let T be an AAT, $V \subseteq$ vertices$_T$, where V is anc-closed and v-data$_T$-closed. If T is version-compatible, then $T|V$ is version-compatible.

Proof: Let $S = T|V$. Note that S is an AAT since V is anc-closed. Let $A \in$ datasteps$_S(x)$.

We must show that label$_S(A) =$ result(x,r), where $r = \langle\langle$v-data$_S(A)$; data$_S\rangle\rangle$.

By definition, label$_S(A) =$ label$_T(A)$.

But T is version-compatible,

$\Rightarrow$ label$_T$(A) = result(x,r'), where
r' = <<v-data$_T$(A); data$_T$>>.

Thus it suffices to show r = r'. But data$_S \subseteq$ data$_T$; thus it suffices to show set equality. It is obvious that r $\subseteq$ r'.

So suppose B $\in$ r', B $\neq$ A,
$\Rightarrow$ B $\in$ visible$_T$(A) $\wedge$ (B,A) $\in$ data$_T$,
$\Rightarrow$ B $\in$ V, since V is v-data$_T$-closed and A $\in$ V,
$\Rightarrow$ B $\in$ visible$_S$(A) $\wedge$ (B,A) $\in$ data$_S$ (since V is anc-closed),
$\Rightarrow$ B $\in$ r. ∎

Lemma 6.3.2.2: Let T $\in \mathcal{R}_a$. Then T is version-compatible.

Proof: Let A $\in$ datasteps$_T$(x). We must show that u (= label$_T$(A)) = result(x,s), where s = <<v-data$_T$(A); data$_T$>>.

Let T = T_0v, where v $\in \mathcal{V}_a$ can be written as $\varphi\pi\psi$, with π = perform A,u.
Let T1 = $T_0\varphi$, and let T2 = $T_0\varphi\pi$.
By precondition Pa.4c, u = result(x,s'), where s' = <<visible$_{T1}$(A,x); data$_{T1}$>>.

Thus it suffices to show s = s', and since data$_{T1} \subseteq$ data$_T$, it suffices to show set equality.

First, let B $\in$ s.
(B,A) $\in$ data$_T$, but A $\in$ datasteps$_{T2}$ $\Rightarrow$ (B,A) $\in$ data$_{T2}$,
$\Rightarrow$ B $\in$ datasteps$_{T2}$,
$\Rightarrow$ B $\in$ datasteps$_{T1}$(x).

By precondition Pa.4b, B $\in$ datasteps$_{T1}$(x) $\Rightarrow$ B $\in$ visible$_{T1}$(A,x) $\vee$ B $\in$ dead$_{T1}$(A,x).

But if B $\in$ dead$_{T1}$(A,x), then B $\in$ dead$_T$(A,x) by Lemma 6.3.1.1.1e,
$\Rightarrow$ B $\notin$ visible$_T$(A), which is a contradiction.

Thus B $\in$ visible$_{T1}$(A,x), $\Rightarrow$ B $\in$ s'.

Conversely, suppose B $\in$ s'. We know B $\neq$ A since A $\notin$ datasteps$_{T1}$.

$(B,A) \in data_{T2} \Rightarrow (B,A) \in data_T$.

$B \in visible_{T1}(A,x) \Rightarrow B \in visible_T(A,x), \Rightarrow B \in s$. ∎

6.3.3 Properties of Aborts Sets in La

In this section we present some properties of aborts sets for reachable trees in La. The first lemma is not strictly a property of aborts sets, but it justifies use of the recursive form of ABORTS in inductive proofs, so we include it here.

Lemma 6.3.3.1: Let $T \in \mathcal{R}_a$, $A \in vertices_T$. Then $A \notin$ v-precedes$^+_T(A)$.

Proof: Let $T = T_0 v$ for some $v \in \mathcal{V}_a$. Suppose $A \in$ v-precedes$^+_T(A)$.
By Lemma 6.3.1.2.11, $A \in committed_T$, and

$$\text{commit } A \underset{v}{\rightarrow} \text{commit } A.$$

But $\underset{v}{\rightarrow}$ is acyclic for $v \in \mathcal{V}_a$, so this is impossible. ∎

Lemma 6.3.3.2: Let $T \in \mathcal{R}_a$, $A \in committed_T$, $(A,B) \in seq$, $A \neq B$, then

$$ABORTS_T(A) \cap desc(B) = \emptyset$$

Proof: Let $T = T_0 v$, for some $v \in \mathcal{V}_a$.

$$ABORTS_T(A) = \bigcup_{C \in vset_T(A)} \text{i-precedes}_T(C) .$$

Let $D \in$ i-precedes$_T(C)$, for some $C \in vset_T(A)$. We show $D \notin desc(B)$. Since $A \in committed_T$, create $D \underset{v}{\rightarrow}$ commit A, by Lemma 6.3.1.2.14.

But if $D \in desc(B)$, then commit A $\underset{v}{\rightarrow}$ create D, by Lemma 6.3.1.2.7. But $\underset{v}{\rightarrow}$ must be acyclic, so we have a contradiction. ∎

Lemma 6.3.3.3: Let $(T,T1) \in \mathcal{R}_a^{(2)}$, and let $A \in committed_T$. Then

$$ABORTS_{T1}(A) \leq ABORTS_T(A)$$

Proof: $ABORTS_{T1}(A) = \bigcup_{B \in vset_{T1}(A)} \text{i-precedes}_{T1}(B)$

By Lemma 6.3.1.1.3e, $\text{vset}_{T1}(A) = \text{vset}_T(A)$

$$\Rightarrow \text{ABORTS}_{T1}(A) = \bigcup_{B \in \text{vset}_T(A)} \text{i-precedes}_{T1}(B)$$

But $A \in \text{committed}_T$, and $\text{v-precedes}_T^+(A) \subseteq \text{committed}_T$ by Lemma 2.3.2,

$\Rightarrow \text{vset}_T(A) \subseteq \text{committed}_T$ by definition of vset_T.

But $B \in \text{committed}_T \Rightarrow \text{i-precedes}_{T1}(B) \leq \text{i-precedes}_T(B)$, by Lemma 6.3.1.1.3f,

$\Rightarrow \text{ABORTS}_{T1}(A) \leq \bigcup_{B \in \text{vset}_T(A)} \text{i-precedes}_T(B)$ (using Lemma 2.2.1.1),

$\Rightarrow \text{ABORTS}_{T1}(A) \leq \text{ABORTS}_T(A)$. ▮

Lemma 6.3.3.4: Let $(T,T1) \in \mathcal{R}_a^{(2)}$, and let $A \in \text{vertices}_T$. Then

$$\text{SEQ-ABORTS}_{T1}(A) \leq \text{SEQ-ABORTS}_T(A)$$

Proof: $\text{SEQ-ABORTS}_{T1}(A) = \text{i-anc-seq}_{T1}(A) \cup \bigcup_{B \in \text{v-anc-seq}_{T1}(A)} \text{ABORTS}_{T1}(B)$

$= \text{i-anc-seq}_T(A) \cup \bigcup_{B \in \text{v-anc-seq}_T(A)} \text{ABORTS}_{T1}(B)$, by Lemma 6.3.1.1.1h.

But $B \in \text{v-anc-seq}_T(A) \Rightarrow B \in \text{committed}_T$,

$\Rightarrow \text{ABORTS}_{T1}(B) \leq \text{ABORTS}_T(B)$, by Lemma 6.3.3.3.

The lemma follows directly using Lemma 2.2.1.1. ▮

6.4 Proof of Possibilities Map for h_{10}

We now return to the task of showing that h_{10} is a possibilities map. First we must prove Lemmas 6.2.4.5, 6.2.4.6, and 6.2.4.7.

We first state an obvious lemma for L1: all reachable AAT's are in ANC-ABORT.

Lemma 6.4.1: Let $T \in \mathcal{R}_1$. Then $T \in$ ANC-ABORT.

Proof: Let $T = T_0 v$, for some $v \in \Upsilon_1$. If $v \neq \Lambda$, then $T = T'e$ for some $e \in \mathcal{E}_1$, $T' \in \text{PRE}_1(e)$, and by the ANC-ABORT precondition for e, $T \in$ ANC-ABORT. If $v = \Lambda$, then $T = T_0$ which is trivially in ANC-ABORT. ▮

We will use this ANC-ABORT property, together with results from La, to prove Lemmas 6.2.4.5, 6.2.4.6, and 6.2.4.7.

Let Ia denote the property of T which is the conjunction of the properties stated in Lemmas 6.3.1.1.2, 6.3.1.1.4, 6.3.2.2, 6.3.3.1, and 6.3.3.2. (Recall that all invariant abbreviations are cross-referenced to lemmas in Appendix I.)

Let Ja denote the pair-property of T which is the conjunction of the pair-properties stated in Lemmas 6.3.1.1.1, 6.3.1.1.3, 6.3.3.3, and 6.3.3.4.

Lemma 6.4.2: Ia is invariant in L1, and Ja is pair-invariant in L1.

Proof: h_{1a} is a possibilities map by Theorem 6.3.1. But h_{1a} fixes T. Since Ia is invariant for T in La, Ia is invariant for T in L1 by Lemma 4.2.4.3.5. Similarly since Ja is pair-invariant for T in La, Ja is pair-invariant for T in L1, by Lemma 4.2.4.3.5. ∎

Let Sa denote the property of event sequences which is the conjunction of the properties stated in Lemmas 6.3.1.2.1 through 6.3.1.2.14.

Lemma 6.4.3: Let v $\in \mathcal{V}_1$. Then Sa holds for v.

Proof: Since h_{1a} is a possibilities map, it is a valid interpretation, by Lemma 4.2.2.5. Thus $h_{1a}(v) \in \mathcal{V}_a$. But h_{1a} is the identity map on events, so $h_{1a}(v) = v$. Since Sa holds for all event sequences in $\mathcal{V}_a$, Sa holds for v. ∎

Now we prove a preliminary lemma for L1, which shows that tracing back the visible precedence relation from any action cannot lead to an ancestor of that action.

Lemma 6.4.4: Let $T \in \mathcal{T}_1$, and let $A \in$ vertices$_T$. Then anc(A) $\cap$ v-precedes$_T^+$(A) $= \emptyset$.

Proof: Suppose B $\in$ anc(A) $\cap$ v-precedes$_T^+$(A).

By Lemma 2.3.2, B $\in$ v-precedes$_T^+$(A) $\Rightarrow$ B $\in$ committed$_T$. By Lemma 6.3.1.1.2f, A $\in$ desc(B) $\Rightarrow$ A $\in$ visible$_T$(B), or A $\in$ dead$_T$(B).

If A $\in$ visible$_T$(B), then by Lemma 6.3.1.2.9, commit A $\xrightarrow[v]{}{}^{*}$ commit B.

But B $\in$ v-precedes$^{+}_{T}$(A), A $\in$ committed$_T$ $\Rightarrow$ commit B $\xrightarrow[v]{}$ commit A, by Lemma 6.3.1.2.11, a contradiction.

If A $\in$ dead$_T$(B), then let C be the lowest (in ancestor order) action in anc(A) $\cap$ v-desc$_T$(B). Clearly C $\in$ vset$_T$(B) $\Rightarrow$ C $\in$ vset$_T$(A), by Lemma 2.3.3a. A $\in$ prop-desc(C) since A $\notin$ visible$_T$(B). Let D = C$\downarrow$A.

But D $\notin$ committed$_T$, since otherwise D would be visible to B, contradicting our choice of C as the *lowest* visible descendant of B which is an ancestor of A.

$\Rightarrow$ D $\in$ aborted$_T$ (by Lemma 6.3.1.1.2c),

$\Rightarrow$ D $\in$ i-child$_T$(C) $\Rightarrow$ D $\in$ ABORTS$_T$(C).

But C $\in$ vset$_T$(A) $\Rightarrow$ ABORTS$_T$(C) $\subseteq$ ABORTS$_T$(A), by Lemma 6.2.1.5,

$\Rightarrow$ D $\in$ ABORTS$_T$(A).

But D $\in$ anc(A), which contradicts T $\in$ ANC-ABORT. ▮

6.4.1 Proof of Lemma 6.2.4.5

Let T $\in \mathfrak{T}_1$, A $\in$ vertices$_T$. Let S = vtree$_T$(A). By Lemma 6.4.4, anc(A) $\cap$ v-precedes$^{+}_{T}$(A) = $\emptyset$,

$\Rightarrow$ prop-anc(A) $\cap$ vset$_T$(A) = $\emptyset$ (since vset$_T$(A) = v-precedes$^{+}_{T}$(A) $\cup$ {A}),

$\Rightarrow$ S is a view tree for A in T, by Lemma 3.5.2. ▮

6.4.2 Proof of Lemma 6.2.4.6

Let T $\in \mathfrak{T}_1$, A $\in$ vertices$_T$. Let S = vtree$_T$(A). By Lemma 6.3.2.2, T is version-compatible.

Let W = vset$_T$(A) $\cup$ prop-anc(A). By Lemmas 6.4.4 and 3.5.2, vtree$_T$(A) = (T|W)//A. W is v-data$_T$-closed by Lemma 2.3.3c. By Lemma 6.3.2.1, T|W is version-compatible. But since backing up proper ancestors of A to active status cannot affect the labels of accesses of S, S is version-compatible. ▮

6.4.3 Proof of Lemma 6.2.4.7

Let $T \in \mathfrak{T}_1$, $A \in \text{vertices}_T$.

We show that if $S = \text{vtree}_T(A)$, then $\text{seq}_S \cup \text{sibling-data}_S$ is acyclic. Let $V = \text{vset}_T(A)$, $W = \text{vset}_T(A) \cup \text{prop-anc}(A)$. By Lemmas 6.4.4 and 3.5.2, $S = (T|W)//A$. Thus $\text{data}_S \subseteq \text{data}_T$, $\text{seq}_S \subseteq \text{seq}_T$. The proof will be by contradiction:

Let $(A_1, A_2, \ldots, A_n)$ be a cycle in $\text{seq}_S \cup \text{sibling-data}_S$ (with $n \geq 2$).
then $(A_1, A_2, \ldots, A_n)$ is a cycle in $\text{seq}_T \cup \text{sibling-data}_T$, and $A_i \in W$.
Let P be the common parent of $\{A_i\}$.

We will use the convention that subscripts are taken modulo n, i.e. we regard $A_{n+1} = A_1$.
First we prove a preliminary lemma:

Lemma 6.4.3.1: If $A \notin \text{desc}(A_i) \cup \text{desc}(A_{i+1})$, then $A_i \in \text{v-precedes}_T^+(A_{i+1})$.

Proof: We show $A_i \in \text{vset}_T(A_{i+1})$. Since $A_i \neq A_{i+1}$, the Lemma follows directly.

$(A_i, A_{i+1}) \in \text{seq}_S$
$\Rightarrow (A_i, A_{i+1}) \in \text{seq}_T$, and $A_i, A_{i+1} \in \text{visible}_T(A)$ (since $\text{vertices}_S \subseteq \text{visible}_T(A)$.)
But $A \notin \text{desc}(A_i) \cup \text{desc}(A_{i+1}) \Rightarrow A_i, A_{i+1} \in \text{visible}_T(P)$,
$\Rightarrow A_i \in \text{v-seq}_T(A_{i+1})$,
$\Rightarrow A_i \in \text{vset}_T(A_{i+1})$.

$(A_i, A_{i+1}) \in \text{sibling-data}_S$
$\Rightarrow \exists a_i \in \text{desc}(A_i), a_{i+1} \in \text{desc}(A_{i+1})$: $(a_i, a_{i+1}) \in \text{data}_T$, and $a_i, a_{i+1} \in \text{visible}_T(A)$ (since $\text{vertices}_S \subseteq \text{visible}_T(A)$.)
But $A \notin \text{desc}(A_i) \cup \text{desc}(A_{i+1}) \Rightarrow a_i, a_{i+1} \in \text{visible}_T(P)$,
$\Rightarrow a_i \in \text{visible}_T(a_{i+1})$, $\Rightarrow a_i \in \text{v-data}_T(a_{i+1})$,
$\Rightarrow A_i \in \text{v-data-anc}_T(a_{i+1})$, $\Rightarrow A_i \in \text{vset}_T(a_{i+1})$.

But $a_{i+1} \in \text{visible}_T(P)$
$\Rightarrow a_{i+1} \in \text{v-desc}_T(A_{i+1})$, $\Rightarrow a_{i+1} \in \text{vset}_T(A_{i+1})$,
$\Rightarrow A_i \in \text{vset}_T(A_{i+1})$. ∎

Proof of Lemma 6.2.4.7 (continued):

Suppose that $A \notin desc(A_i)$ $\forall i$; then by Lemma 6.4.3.1, $A_i \in$ v-precedes$^+_T(A_{i+1})$ $\forall i$,

$\Rightarrow$ $A_i \in$ v-precedes$^+_T(A_i)$ (since the A_i form a cycle), which contradicts Lemma 6.3.3.1.

Thus $A \in desc(A_i)$, for some i. Assume without loss of generality that $A \in desc(A_n)$.

Since $n \geq 2$, $A_1 \neq A_n$. But $A_1 \in$ vertices$_S$, and $A_1 \notin anc(A)$,

$\Rightarrow$ $A_1 \in$ vprecedes$^+_T(A)$.

$A_1 \in$ vprecedes$^+_T(A)$

$\Rightarrow$ ABORTS$_T(A_1) \subseteq$ ABORTS$_T(A)$, by Lemma 6.2.1.5.

Since $(A_i,A_{i+1}) \in$ seq$_S$ $\cup$ sibling-data$_S$, we have two following cases:

1. $(A_n,A_1) \in$ seq$_S$
2. $(A_n,A_1) \in$ sibling-data$_S$

Case 1: $(A_n,A_1) \in$ seq $\Rightarrow$ $A_n \in$ done$_T$, by Lemma 6.3.1.1.2b.

If $A_n \in$ aborted$_T$, then $A_n \in$ i-anc-seq$_T(A_1)$, $\Rightarrow$ $A_n \in$ ABORTS$_T(A_1)$,

$\Rightarrow$ $A_n \in$ ABORTS$_T(A)$, which contradicts $T \in$ ANC-ABORT.

If $A_n \in$ committed$_T$, then $A_n \in$ v-anc-seq$_T(A_1)$, $\Rightarrow$ $A_n \in$ v-precedes$_T(A_1)$,

$\Rightarrow$ $A_n \in$ v-precedes$^+_T(A)$, which contradicts Lemma 6.4.4.

Case 2: $(A_n,A_1) \in$ sibling-data$_S$

$\Rightarrow$ $\exists b_n \in desc(A_n), b_1 \in desc(A_1)$: $(b_n, b_1) \in$ data$_T$.

$b_1 \in$ visible$_T(A)$ $\Rightarrow$ $b_1 \in$ visible$_T(P)$, $\Rightarrow$ $b_1 \in$ v-desc$_T(A_1)$, $\Rightarrow$ $b_1 \in$ vset$_T(A_1)$,

$\Rightarrow$ $b_1 \in$ v-precedes$^+_T(A)$,

$\Rightarrow$ ABORTS$_T(b_1) \subseteq$ ABORTS$_T(A)$.

Case 2a: $b_n \in$ visible$_T(b_1)$,

$\Rightarrow$ $b_n \in$ v-data$_T(b_1)$, $\Rightarrow$ $A_n \in$ v-data-anc$_T(b_1)$, $\Rightarrow$ $A_n \in$ v-precedes$_T(b_1)$,

$\Rightarrow$ $A_n \in$ v-precedes$^+_T(A)$, contradicting Lemma 6.4.4.

Case 2b: $b_n \notin \text{visible}_T(b_1)$ ·
$\Rightarrow b_n \in \text{i-data}_T(b_1)$ (See Fig. 6.1.)

Let $B = \text{crucial}_T(b_n)$. By Lemma 6.3.1.1.2g, B is defined, and $B \in \text{desc}(A_n)$.
$B \in \text{i-data-anc}_T(b_1) \Rightarrow B \in \text{ABORTS}_T(b_1), \Rightarrow B \in \text{ABORTS}_T(A)$.

But $B \in \text{anc}(A)$, since $b_n \in \text{visible}_T(A)$, contradicting $T \in$ ANC-ABORT. ▮

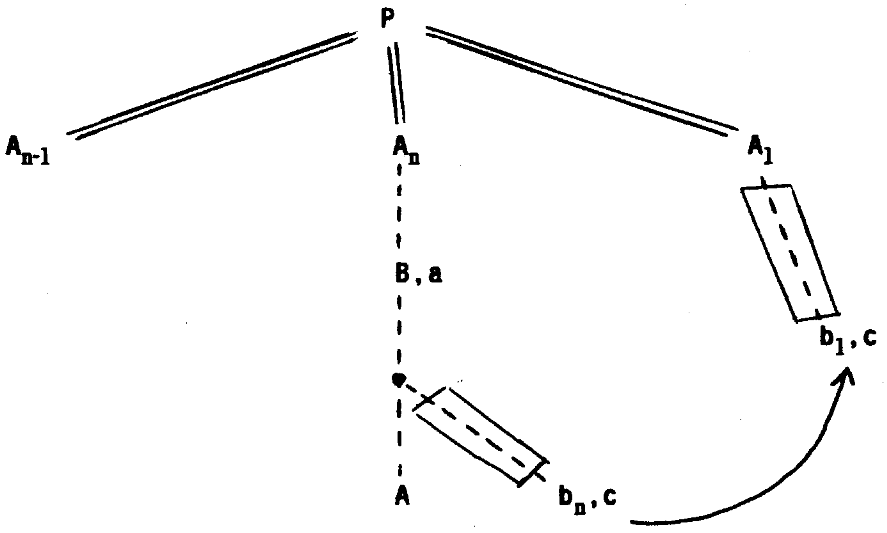
6.4.4 Proof that h_{10} is a Possibilities Map

We now have all the facts needed to show that h_{10} is a possibilities map:

Theorem 6.4.4.1: h_{10} is a possibilities map.

Proof: h_{10} preserves initial states by Lemma 6.2.4.1. h_{10} preserves transitions by Lemma 6.2.4.2. We have proven Lemmas 6.2.4.5, 6.2.4.6, and 6.2.4.7; thus every reachable state in L1 is view-serializable (Theorem 6.2.4.4). Thus h_{10} preserves preconditions by Lemma 6.2.4.3. By Lemma 4.2.2.6, h_{10} is a possibilities map. ▮

Fig. 6.1. Case 2b, Lemma 6.2.4.7



6.5 Level 2 Algebra and Mapping h_{21}

At Level 2 we replace the ANC-ABORT condition with a precondition on perform events. Otherwise everything in Level 2 is identical to Level 1.

$$L2 = (\mathcal{E}_2, \Sigma_2, \sigma_2, \tau_2)$$

$\mathcal{E}_2 = \mathcal{E}_1 = \{\text{create } A, \text{ commit } A, \text{ abort } A, \text{ perform } A,u\}$.

$\Sigma_2 = \Sigma_1$, the set of all augmented action trees.

$\sigma_2 = \sigma_1 = T_0$.

τ_2, the transition relation, is obtained by deleting the ANC-ABORT preconditions P1.1d, P1.2c, P1.3b, and P1.4d, and by inserting a new precondition for perform events:

(P2.4d) $B \in \text{visible}_T(A,x) \Rightarrow \text{anc}(A) \cap \text{ABORTS}_T(A{\downarrow}B) = \emptyset$

We define the trivial mapping h_{2a}: L2 → La as the identity map on states and events.

Theorem 6.5.1: h_{2a} is a possibilities map.

Proof: Because the ANC-ABORT conditions do not appear in algebra La, every precondition at Level A also appears at Level 2 (in addition, Level 2 has precondition P2.4d). Thus h_{2a} preserves preconditions. All transitions are identical in La and L2; thus h_{2a} preserves transitions. Initial states are identical in L2 and La; thus h_{2a} preserves initial states. By Lemma 4.2.2.6, h_{2a} is a possibilities map. ∎

Since h_{2a}.fixes T, all invariants (and pair-invariants) for La are invariant (or pair-invariant) for L2:

Lemma 6.5.2: Ia is invariant in L2, and Ja is pair-invariant in L2.

Proof: Since h_{2a} is a possibilities map which fixes T, and Ia is invariant for T in La, Ia is invariant for T in L2 by Lemma 4.2.4.3.5. Similarly since Ja is pair-invariant for T in La, Ja is pair-invariant for T in L2, by Lemma 4.2.4.3.5. ∎

6.5.1 Specification of Mapping h_{21}

We define the trivial mapping h_{21}: L2 → L1 as the identity map on states and events. We must show that this mapping is a possibilities map.

Lemma 6.5.1.1: h_{21} preserves initial states.

Proof: Trivial, since $\sigma_2 = \sigma_1$. ▮

Lemma 6.5.1.2: h_{21} preserves transitions.

Proof: Trivial, since all transitions are identical in L2 and L1. ▮

We must also show that h_{21} preserves preconditions. We use the following lemma to reduce this problem to the ANC-ABORT condition on reachable states in L2:

Lemma 6.5.1.3: Suppose that for all $T \in \mathfrak{R}_2$, $T \in$ ANC-ABORT. Then h_{21} preserves preconditions.

Proof: It is obvious that h_{21} preserves all preconditions except for the ANC-ABORT conditions, since all other preconditions appear at Level 2. We must verify that the ANC-ABORT conditions hold; these conditions state that the *next state* is in ANC-ABORT, i.e.

$$T \in PRE_2(e) \cap \mathfrak{R}_2, h_{21}(T) \in \mathfrak{R}_1 \Rightarrow h_{21}(T)h_{21}(e) \in \text{ANC-ABORT}$$

But $h_{21}(T)h_{21}(e)$ is just Te, since h_{21} is the identity mapping. Thus we will must show

$$T \in PRE_2(e) \cap \mathfrak{R}_2, h_{21}(T) \in \mathfrak{R}_1 \Rightarrow Te \in \text{ANC-ABORT}.$$

But $T \in PRE_2(e) \cap \mathfrak{R}_2 \Rightarrow Te \in \mathfrak{R}_2$. Thus it suffices to show that all reachable states in L2 are in ANC-ABORT. ▮

Our main result for L2 is thus that all reachable states are in ANC-ABORT, which will imply that h_{21} preserves preconditions (and is thus a possibilities map):

Lemma 6.5.1.4: Let $T \in \mathcal{R}_2$. Then $T \in$ ANC-ABORT.

Proof: Take $A \in \text{vertices}_T$. We show $\text{anc}(A) \cap \text{ABORTS}_T(A) = \emptyset$.

The proof uses induction based on the recursive form of ABORTS:

$$\text{ABORTS}_T(A) = \text{i-precedes}_T(A) \cup \bigcup_{B \in \text{v-precedes}_T(A)} \text{ABORTS}_T(B)$$

Recall that by Lemma 6.5.2, we can use any results from Ia or Ja since we have shown that these properties are invariant n L2.

Thus Lemma 6.3.3.1 (in Ia) justifies the use of the inductive proof method.

Assume the Lemma holds for all $B \in \text{v-precedes}_T(A)$: $\text{anc}(B) \cap \text{ABORTS}_T(B) = \emptyset$.
First we show $\text{i-precedes}_T(A) \cap \text{anc}(A) = \emptyset$:

$B \in \text{i-anc-seq}_T(A) \Rightarrow (B,A') \in \text{seq}$, for some $A' \in \text{anc}(A)$, $B \neq A'$,
$\Rightarrow B \notin \text{anc}(A)$.

$B \in \text{i-child}_T(A) \Rightarrow B \in \text{children}(A)$,
$\Rightarrow B \notin \text{anc}(A)$.

$B \in \text{i-data-anc}_T(A) \Rightarrow \exists B' \in \text{i-data}_T(A)$: $B = \text{crucial}_T(B')$.
But by Lemma 6.3.1.1.2g, $\text{crucial}_T(B') \in \text{desc}(A \downarrow B)$
$\Rightarrow B \notin \text{anc}(A)$.

Now we show $B \in \text{v-precedes}_T(A) \Rightarrow \text{anc}(A) \cap \text{ABORTS}_T(B) = \emptyset$:

a. $B \in \text{v-anc-seq}_T(A) \Rightarrow (B,A') \in \text{seq}$, for some $A' \in \text{anc}(A)$, $B \neq A'$,
$\Rightarrow \text{ABORTS}_T(B) \cap \text{desc}(A') = \emptyset$, by Lemma 6.3.3.2.
But by Induction Hypothesis, $\text{ABORTS}_T(B) \cap \text{anc}(B) = \emptyset$.
But $\text{anc}(B) = \{B\} \cup \text{proper-anc}(A')$, since $(B,A') \in \text{siblings}$,
$\Rightarrow \text{anc}(A) \subseteq \text{desc}(A') \cup \text{anc}(B)$,
$\Rightarrow \text{ABORTS}_T(B) \cap \text{anc}(A) = \emptyset$.

b. $B \in \text{v-child}_T(A) \Rightarrow \text{ABORTS}_T(B) \cap \text{anc}(B) = \emptyset$, by Induction Hypothesis.
But $B \in \text{children}(A) \Rightarrow \text{anc}(A) \subseteq \text{anc}(B)$,
$\Rightarrow \text{ABORTS}_T(B) \cap \text{anc}(A) = \emptyset$.

c. $B \in$ v-data-anc$_T(A) \Rightarrow \exists B' \in$ v-data$_T(A)$: $B = A{\downarrow}B'$,
$\Rightarrow A \in$ datasteps$_T$.

Let $T = T_0v$, where $v \in \mathcal{T}_2$ can be written as $\varphi\pi\psi$, with π = perform A,u.
Let $T1 = T_0\varphi$, and let $T2 = T_0\varphi\pi$.

Let A,B' $\in$ datasteps(x).
By Lemma 6.3.1.1.1c, $(B',A) \in$ data$_{T2}$,
$\Rightarrow B' \in$ datasteps$_{T1}(x)$.
By precondition P2.4b, $B' \in$ visible$_{T1}(A,x) \vee B' \in$ dead$_{T1}(A,x)$.

$B' \in$ v-data$_T(A) \Rightarrow B' \in$ visible$_{T1}(A,x)$,
$\Rightarrow$ anc(A) $\cap$ ABORTS$_{T1}(A{\downarrow}B') = \emptyset$, by precondition P2.4d (the orphan detection precondition),
$\Rightarrow$ anc(A) $\cap$ ABORTS$_{T1}(B) = \emptyset$.
But ABORTS$_T(B) \leq$ ABORTS$_{T1}(B)$, by Lemma 6.3.3.3 (since $B \in$ committed$_{T1}$),
$\Rightarrow$ ABORTS$_T(B) \cap$ anc(A) $= \emptyset$, by Lemma 2.2.1.1d. ∎

6.5.2 Proof that h_{21} is a Possibilities Map

We now have all the facts needed to show that h_{21} is a possibilities map:

Theorem 6.5.2.1: h_{21} is a possibilities map.

Proof: h_{21} preserves initial states by Lemma 6.5.1.1. h_{21} preserves transitions by Lemma 6.5.1.2. Since we have shown that all reachable states in L2 are in ANC-ABORT (Lemma 6.5.1.4), h_{21} preserves preconditions by Lemma 6.5.1.3. By Lemma 4.2.2.6, h_{21} is a possibilities map. ∎

7. Partially Localized Model

In a distributed event-state algebra all preconditions for events are localized to the nodes at which those events occur (or to the message buffer). The Level 2 model is defined in terms of a single *global state*, the global AAT. As we move towards a distributed model, we partition the state into distinct components, and we attempt to localize preconditions to these components. At Level 3, we define an abstract set of *locations*, and we give each location a (local) state. This local state will consist of a UAS at each location, and an ordering on datasteps at each object. (The data ordering in an AAT is already "localized," since data_T individually orders datasteps at each object.)

These locations are simply containers for information; they need not correspond directly to physical locations (nodes) at the lower levels. In a later chapter we will construct a mapping from a *distributed* model where state is partitioned among *nodes*, to this *localized* model where state is partitioned among *locations*. Essentially several abstract locations can reside at a single physical node. One advantage of using abstract locations at this higher level is that we need not be concerned with how information is physically distributed.

We can think of locations as "abstract nodes." We will consider each action and object to be a separate location; the information at these locations will represent the view at that action or object. It will be convenient to allow other (unspecified) locations as well. The events at this level will be either "local steps," which are conceptually local to a particular location, or "communications steps," which transfer information from one location to another. Transfer of information is instantaneous (i.e. there is no analog to the message buffer at this level). (We will show later that we can model communications delays by regarding "message slots" in the message buffer as locations. Thus we do not specify the complete set of "locations" at this level; locations can be viewed abstractly as *any* information holders.)

We show that it is straightforward to localize all preconditions except for the orphan detection condition (precondition P2.4d: $B \in \text{visible}_T(A,x) \Rightarrow \text{anc}(A) \cap \text{ABORTS}_T(A{\downarrow}B) = \emptyset$).

7.1 Level 3 Algebra

$L3 = (\mathcal{E}_3, \Sigma_3, \sigma_3, \tau_3)$

<u>State Space:</u>

Let tloc (the "tree locations") $=$ (act - $\{U\}$ - accesses) $\cup$ obj.

Let loc be a set of "locations," where tloc $\subseteq$ loc.

(We exclude U from tloc because it is a virtual action; thus we associate no information with it directly. Also, we exclude accesses from tloc because we regard them as being coupled to their objects for information.)

$\Sigma_3 = \{\langle T,L \rangle\}$, where the components are

T - the "global state", an augmented action tree (as in L2)
L - the "local state", where $L\colon loc \rightarrow UAS$

Notation

If "prop" is some property (function, relation, etc.) defined on UAS, then we denote $prop_{L(\alpha)}$ by "$@prop_L[\alpha]$" (for example, $visible_{L(\alpha)}(A,x) = @visible_L[\alpha](A,x)$).
The "@" symbol flags components of the *local* state (as opposed to components of the global AAT). We also use the "@" symbol to distinguish *communications* events from "local" events, since the communications events only affect the local state.

We further abbreviate by writing $@prop_L[A]$ for $@prop_L[x]$ when $A \in accesses(x)$

Initial State:

$\sigma_3 = \langle T_0, L_0 \rangle$

T_0 - the trivial AAT, as in L2,
$L_0(\alpha) = T_u$ - the trivial UAS, $\forall \alpha \in loc$.

Events:

Events create, commit, abort, and perform are localized to individual locations (except for the orphan detection precondition). We regard an action as being created at the location of its *creator*, and committed or aborted at its own location (or the location of its object if it is an access). (Recall that for $A \neq U$, creator(A) = parent(A) unless $A \in top$, and creator(A) = A for $A \in top$.)

In addition to the "local" events (create, commit, abort, perform), we introducc "communications events" to move information from one location to another. The "source" of information is arbitrary for each event: communications events are parameterized by a single location: the destination of the information transfer. At lower levels we will parameterize communications events by the sender of information as well.

The communications events are as follows:

@create[α] A	-- create action A at location α
@commit[α] A	-- commit action A at location α
@abort[α] A	-- abort action A at location α

The transition relation is defined so that each communications event is idempotent, i.e., the effect of a communications event which occurs multiple times is the same as the effect of this event occurring a single time. Idempotency "filters out" duplicate communications events.

<u>Transition Relation:</u>

Let $e \in \mathcal{E}_3$, $\langle T,L\rangle \in \Sigma_3$, $\langle T,L\rangle e = \langle T1,L1\rangle$.

1. <u>create A</u> $(A \in act - \{U\})$

PRECONDITIONS:

a. $A \notin$ @vertices$_L$[creator(A)]

b. parent(A) $\in$ @active$_L$[creator(A)]

c. $(B,A) \in seq$, $B \neq A \Rightarrow B \in$ @done$_L$[creator(A)]

TRANSITIONS:

a. vertices$_{T1}$ $\leftarrow$ vertices$_T$ $\cup \{A\}$

b. status$_{T1}$(A) $\leftarrow$ 'active'

c. @vertices$_{L1}$[creator(A)] $\leftarrow$ @vertices$_L$[creator(A)] $\cup \{A\}$

d. $@status_{L1}[creator(A)](A) \leftarrow$ 'active'

2. commit A $(A \in act - \{U\} - accesses)$

PRECONDITIONS:

a. $A \in @active_L[A]$

b. $@children_L[A](A) \subseteq @done_L[A]$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'committed'

b. $@status_{L1}[A](A) \leftarrow$ 'committed'

3. abort A $(A \in act - \{U\})$

PRECONDITIONS:

a. $A \in @active_L[A]$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'aborted'

b. $@status_{L1}[A](A) \leftarrow$ 'aborted'

4. perform A.u $(A \in accesses(x), u \in values(x))$

PRECONDITIONS:

a. $A \in @active_L[x]$

b. $B \in @datasteps_L[x](x) \Rightarrow B \in @visible_L[x](A,x) \lor B \in @dead_L[x](A,x)$

c. $u = result(x,s)$, where $s = \langle\langle @visible_L[x](A,x); O(x) \rangle\rangle$, and $O = order(T)$.

d. $B \in @visible_L[x](A,x) \Rightarrow anc(A) \cap ABORTS_T(A \downarrow B) = \emptyset$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'committed'

b. $@\text{status}_{L1}[x](A) \leftarrow$ 'committed'

c. $\text{label}_{T1}(A) \leftarrow u$

d. $\text{data}_{T1} \leftarrow \text{data}_T \cup \{(B,A): B \in \text{datasteps}_T(x)\} \cup \{(A,A)\}$

5. $\underline{@\text{create}[\alpha]\ A}$ $\quad (A \in \text{act} - \{U\}, \alpha \in \text{loc})$

PRECONDITIONS:

a. $A \in @\text{vertices}_L[\beta]$, for some $\beta \in \text{loc}$

TRANSITIONS:

a. $@\text{vertices}_{L1}[\alpha] \leftarrow @\text{vertices}_L[\alpha] \cup \{A\}$

b. $A \notin @\text{vertices}_L[\alpha] \Rightarrow @\text{status}_{L1}[\alpha](A) \leftarrow$ 'active'

6. $\underline{@\text{commit}[\alpha]\ A}$ $\quad (A \in \text{act} - \{U\}, \alpha \in \text{loc})$

PRECONDITIONS:

a. $A \in @\text{committed}_L[\beta]$, for some $\beta \in \text{loc}$

TRANSITIONS:

a. $@\text{vertices}_{L1}[\alpha] \leftarrow @\text{vertices}_L[\alpha] \cup \{A\}$

b. $@\text{status}_{L1}[\alpha](A) \leftarrow$ 'committed'

7. $\underline{@\text{abort}[\alpha]\ A}$ $\quad (A \in \text{act} - \{U\}, \alpha \in \text{loc})$

PRECONDITIONS:

a. $A \in @\text{aborted}_L[\beta]$, for some $\beta \in \text{loc}$

TRANSITIONS:

a. $@\text{vertices}_{L1}[\alpha] \leftarrow @\text{vertices}_L[\alpha] \cup \{A\}$

b. $@\text{status}_{L1}[\alpha](A) \leftarrow$ 'aborted'

7.2 Specification of Mapping h_{32}

We define a (single-state) mapping from L3 to L2, h_{32}: L3 → L2. (We abbreviate "h_{32}" as "h" in this chapter.)

State Mapping

$h: \Sigma_3 \to \Sigma_2$ is defined by $h(\langle T,L\rangle) = T, \quad \forall \langle T,L\rangle \in \Sigma_3$. Thus h fixes T.

Event Mapping

$h: \mathcal{E}_3 \to \mathcal{E}_2^*$ is defined by

h: create A → create A
commit A → commit A
abort A → abort A
perform A,u → perform A,u

@create[α] A → Λ
@commit[α] A → Λ
@abort[α] A → Λ

7.3 Level 3 Invariants

The following simple pair-invariants are analogous to the Level Λ pair-invariants from Lemma 6.3.1.1.1:

Lemma 7.3.1: Let $(\langle T,L\rangle,\langle T1,L1\rangle) \in \mathfrak{R}_3^{(2)}$. Then the following pair-invariants hold (Let $\alpha \in$ loc, x ∈ obj, A,B ∈ act):

a. $@\text{vertices}_L[\alpha] \subseteq @\text{vertices}_{L1}[\alpha]$, $@\text{committed}_L[\alpha] \subseteq @\text{committed}_{L1}[\alpha]$, $@\text{aborted}_L[\alpha] \subseteq @\text{aborted}_{L1}[\alpha]$, $@\text{done}_L[\alpha] \subseteq @\text{done}_{L1}[\alpha]$

b. B is dead in $L(\alpha) \Rightarrow$ B is dead in $L1(\alpha)$; B is live in $L1(\alpha) \Rightarrow$ B is live in $L(\alpha)$

c. $@\text{visible}_L[\alpha](\Lambda) \subseteq @\text{visible}_{L1}[\alpha](A)$, $\quad @\text{dead}_L[\alpha](\Lambda) \subseteq @\text{dead}_L[\alpha](A)$

Proof: Straightforward. ∎

The following invariants relate local states to the global state. Essentially each local state represents a "partial view" of the true global state. We show these invariants relative to mapping h: Since h fixes T, all invariants and pair-invariants for T in L2 can be applied to the proofs (by Lemma 4.2.4.3.6). Recall that we have shown in Lemma 6.5.2 that invariants Ia and Ja from Level A are invariant in L2.

Lemma 7.3.2: Let $\langle T,L\rangle \in \mathcal{B}_3$. Then the following are invariant relative to h. (Let $\alpha \in$ loc, $x \in$ obj, $A,B \in$ act):

a. $A \in \mathrm{vertices}_T - \{U\} \Leftrightarrow A \in \mathrm{@vertices}_L[\mathrm{creator}(A)]$;
$U \in \mathrm{active}_T \wedge U \in \mathrm{@active}_L[\alpha]$ $(\forall \alpha \in \mathrm{tloc})$

b. $A \in \mathrm{committed}_T \Leftrightarrow A \in \mathrm{@committed}_L[A]$

c. $A \in \mathrm{aborted}_T \Leftrightarrow A \in \mathrm{@aborted}_L[A]$

d. $A \in \mathrm{done}_T \Leftrightarrow A \in \mathrm{@done}_L[A]$

e. $A \in \mathrm{datasteps}_T(x) \Leftrightarrow A \in \mathrm{@datasteps}_L[x](x)$

f. $\mathrm{@vertices}_L[\alpha] \subseteq \mathrm{vertices}_T$, $\mathrm{@committed}_L[\alpha] \subseteq \mathrm{committed}_T$, $\mathrm{@aborted}_L[\alpha] \subseteq \mathrm{aborted}_T$, $\mathrm{@done}_L[\alpha] \subseteq \mathrm{done}_T$
(Note: $\mathrm{@active}_L[\alpha] \subseteq \mathrm{active}_T$ does *not* necessarily hold.)

g. $A \in \mathrm{@active}_L[A] \Rightarrow A \in \mathrm{active}_T$
(Note: not necessarily conversely)

h. $B \in \mathrm{@visible}_L[\alpha](A) \Rightarrow B \in \mathrm{visible}_T(A)$

i. $B \in \mathrm{@dead}_L[\alpha](A) \Rightarrow B \in \mathrm{dead}_T(A)$

j. $(B,A) \in \mathrm{data}_T$ $(A,B \in \mathrm{accesses}(x)) \Rightarrow$
$B \in \mathrm{visible}_T(A) \Leftrightarrow B \in \mathrm{@visible}_L[x](A)$

k. $(B,A) \in \mathrm{data}_T$ $(A,B \in \mathrm{accesses}(x)) \Rightarrow$
$B \in \mathrm{dead}_T(A) \Leftrightarrow B \in \mathrm{@dead}_L[x](A)$

l. $(B,A) \in \mathrm{data}_T$ $(A,B \in \mathrm{accesses}(x)) \Rightarrow$
$B \in \mathrm{@dead}_L[x](A) \Rightarrow \{\mathrm{crucial}_T(B)\} \leq \mathrm{@aborted}_L[x]$

Proof:

a. If $A \neq U$, $A \in \text{vertices}_T$, then there must have been an event create A, which also has the effect of placing A in $@\text{vertices}_L[\text{creator}(A)]$. Using Lemma 7.3.1a, we conclude that $A \in \text{vertices}_T \Rightarrow A \in @\text{vertices}_L[\text{creator}(A)]$.

Conversely, if $A \in @\text{vertices}_L[\text{creator}(A)]$ then there must have been an event create A, or $@\text{create}[\text{creator}(A)]$ A. Consider the first such event. If it is create A then $A \in \text{vertices}_T$. Now suppose there were an event $@\text{create}[\alpha]$ A, for some α, that preceded create A. Let e be the first such event, and let the state immediately before the execution of e be $\langle T1,L1\rangle$. By the precondition for e, $A \in @\text{vertices}_{L1}[\beta]$ for some β. But if $A \in @\text{vertices}_{L1}[\beta]$, then either $\beta = \text{creator}(A)$ and create A precedes e, or an event $f = @\text{create}[\beta]$ A precedes e. Both cases contradict our choice of e.

$U \in \text{active}_T$ by Lemma 6.3.1.1.2d. To see that $U \in @\text{active}_L[\alpha]$, note that $U \in @\text{active}_{L_0}[\alpha]$, but no event can change U's status.

b. Similar to (a).

c. Similar to (a).

d. Follows directly from (b) and (c).

e. $A \in \text{datasteps}_T(x) \Leftrightarrow A \in \text{committed}_T$,
$\Leftrightarrow A \in @\text{committed}_L[x]$ by (b),
$\Leftrightarrow A \in @\text{datasteps}_L[x](x)$.

f. We argue $@\text{vertices}_L[\alpha] \subseteq \text{vertices}_T$; the other cases are similar: If $\alpha = \text{creator}(A)$, then the result follows directly from (a). Otherwise we can show $@\text{vertices}_L[\alpha] \subseteq \text{vertices}_T$ by induction on the number of events in a valid sequence generating $\langle T,L\rangle$. In the initial state $@\text{vertices}_L[\alpha] = \text{vertices}_T = \{U\}$. But $\text{vertices}_L[\alpha]$ can only increase when an event $@\text{create}[\alpha]$ A occurs, which requires as precondition $A \in @\text{vertices}_{L1}[\beta]$ for some β (where $\langle T1,L1\rangle$ is the state before this event occurs). By induction hypothesis, $A \in @\text{vertices}_{L1}[\beta] \Rightarrow A \in \text{vertices}_{T1} \Rightarrow A \in \text{vertices}_T$.

g. $A \in @\text{active}_L[A] \Rightarrow A \in \text{vertices}_T$ from (f). If $A \in \text{done}_T$, then $A \in @\text{done}_L[A]$ by (d) -- a contradiction. Thus $A \in \text{active}_T$.

h. $B \in @\text{visible}_L[\alpha](A) \Rightarrow \text{anc}(A) \cap \text{prop-desc}(\text{lca}(A,B)) \subseteq @\text{committed}_L[\alpha]$.
But $@\text{committed}_L[\alpha] \subseteq \text{committed}_T$ by (f),
$\Rightarrow B \in \text{visible}_T(A)$.

i. $B \in @\text{dead}_L[\alpha](A) \Rightarrow \text{anc}(A) \cap \text{prop-desc}(\text{lca}(A,B)) \cap @\text{aborted}_L[\alpha] \neq \emptyset$.
But $@\text{aborted}_L[\alpha] \subseteq \text{aborted}_T$ by (f),
$\Rightarrow B \in \text{dead}_T(A)$.

j. $B \in @visible_L[x](A) \Rightarrow B \in visible_T(A)$ by (h).
$B \in visible_T(A) \Rightarrow B \in visible_T(A,x)$,
$\Rightarrow B \in @datasteps_L[x](x)$. Assume $B \neq A$ (otherwise the result is obvious).

Let $\langle T,L\rangle = \sigma_3 v$, $v \in \Upsilon_3$.
Let $v = \varphi\varpi\psi$, where $\varpi =$ perform A,u, and let $\langle T1,L1\rangle = \sigma_3\varphi$.

$(B,A) \in data_T \Rightarrow (B,A) \in data_{T1}$, by Lemma 6.3.1.1.1c, $\Rightarrow B \in datasteps_{T1}$,
$\Rightarrow B \in @datasteps_{L1}[x](x)$, by (e),
$\Rightarrow B \in @visible_{L1}[x](A,x) \vee B \in @dead_{L1}[x](A,x)$ by P3.4b.

But $B \in @dead_{L1}[x](A,x) \Rightarrow B \in dead_{T1}(A,x)$ (by (i)), $\Rightarrow B \in dead_T(A,x)$ -- a contradiction. Thus $B \in @visible_{L1}[x](A,x)$,
$\Rightarrow B \in @visible_L[x](A,x)$ (using Lemma 7.3.1c).

k. Similar to (j) above.

l. $B \in @dead_L[x](A,x) \Rightarrow anc(B) \cap @aborted_L[x] \neq \emptyset$,
$\Rightarrow @crucial_L[x](B)$ is defined.
But $anc(B) \cap @aborted_L[x] \subseteq anc(B) \cap aborted_T$, by (f),
$\Rightarrow crucial_T(B) \in desc(@crucial_L[x](B))$,
$\Rightarrow \{crucial_T(B)\} \leq @aborted_L[x]$.

7.4 Proof of Possibilities Map for h_{32}

We now show that h is a possibilities map. Let I3 denote the conjunction of all properties in Lemma 7.3.2. We will show that h is a possibilities map relative to I3.

Lemma 7.4.1: h preserves initial states.

Proof: Immediate since $h(\langle T_0,L_0\rangle) = T_0$. ∎

Lemma 7.4.2: h preserves transitions relative to I3.

Proof: We must show that if $\langle T,L\rangle \in PRE_3(e) \cap \mathfrak{R}_3 \cap I3$, and $h(\langle T,L\rangle) \in PRE_2(h(e)) \cap \mathfrak{R}_2$, then $h(\langle T,L\rangle e) = h(\langle T,L\rangle)h(e)$.

But $h(\langle T,L\rangle) = T$, so we must show the following:

If $\langle T,L\rangle \in PRE_3(e) \cap \mathcal{R}_3 \cap I3$, and $T \in PRE_2(h(e)) \cap \mathcal{R}_2$, and $\langle T,L\rangle e = \langle T1,L1\rangle$, then $T1 = (T)h(e)$.

For the communications steps in L3 (e = @create, @commit, @abort), T is not altered, and $h(e) = \Lambda$, so $T1 = T = (T)h(e)$.

For the local steps (e = create, commit, abort, perform), it is easily verified by inspection that the effects of events on T are identical in L2 and L3. But $h(e) = e$, so $T1 = (T)h(e) = (T)e$. ∎

Lemma 7.4.3: h preserves preconditions relative to I3.

Proof: We must show that if $\langle T,L\rangle \in PRE_3(e) \cap \mathcal{R}_3 \cap I3$, and $h(\langle T,L\rangle) \in \mathcal{R}_2$, then $h(\langle T,L\rangle) \in PRE_2(h(e))$.

Since $h(\langle T,L\rangle) = T$, we show
$\langle T,L\rangle \in PRE_3(e) \cap \mathcal{R}_3 \cap I3, T \in \mathcal{R}_2 \Rightarrow T \in PRE_2(h(e))$.

For communications steps, e, $h(e) = \Lambda$, and preservation of preconditions follows vacuously. For local steps, $h(e) = e$. We prove preservation of preconditions for each local step in turn:

1. create A

   a. P3.1a $\Rightarrow A \notin$ @vertices$_L$[creator(A)],
      $\Rightarrow A \notin$ vertices$_T$, by Lemma 7.3.2a.

   b. If $A \in$ top, then parent(A) = U, $U \in$ active$_T$ by Lemma 7.3.2a.
      Otherwise creator(A) = parent(A),
      $\Rightarrow$ parent(A) $\in$ @active$_L$[parent(A)] by P3.1b,
      $\Rightarrow$ parent(A) $\in$ active$_T$ by Lemma 7.3.2g.

   c. $(B,A) \in$ seq, $B \neq A \Rightarrow B \in$ @done$_L$[creator(A)] by P3.1c,
      $\Rightarrow B \in$ done$_T$ by Lemma 7.3.2g.

2. commit A

a. P3.2a $\Rightarrow$ $A \in$ @active$_L$[A],
$\Rightarrow$ $A \in \text{active}_T$ by Lemma 7.3.2g.

b. Let $B \in \text{children}_T(A)$. $A \neq U \Rightarrow B \notin \text{top}$,
$\Rightarrow$ creator(B) = A,
$\Rightarrow$ $B \in$ @vertices$_L$[A] by Lemma 7.3.2a,
$\Rightarrow$ $B \in$ @children$_L$[A](A),
$\Rightarrow$ $B \in$ @done$_L$[A] by P3.2b,
$\Rightarrow$ $B \in \text{done}_T$ by Lemma 7.3.2f.

3. abort A

a. P3.3a $\Rightarrow$ $A \in$ @active$_L$[A],
$\Rightarrow$ $A \in \text{active}_T$ by Lemma 7.3.2g.

4. perform A,u

a. P3.4a $\Rightarrow$ $A \in$ @active$_L$[x],
$\Rightarrow$ $A \in$ @active$_L$[A],
$\Rightarrow$ $A \in \text{active}_T$ by Lemma 7.3.2g.

b. $B \in \text{datasteps}_T(x)$ $\Rightarrow$ $B \in$ @datasteps$_L$[x](x) by Lemma 7.3.2e,
$\Rightarrow$ $B \in$ @visible$_L$[x](A,x) $\vee$ $B \in$ @dead$_L$[x](A,x) by P3.4b.

$B \in$ @visible$_L$[x](A,x) $\Rightarrow$ $B \in \text{visible}_T(A,x)$ by Lemma 7.3.2h.
$B \in$ @dead$_L$[x](A,x) $\Rightarrow$ $B \in \text{dead}_T(A,x)$ by Lemma 7.3.2i.

c. P3.4c $\Rightarrow$ $u = \text{result}(x,s)$, where $s = \langle\langle$@visible$_L$[x](A,x); $\mathcal{O}(x)\rangle\rangle$, and $\mathcal{O} = \text{order}(T)$. We must show $s = s'$, where $s' = \langle\langle \text{visible}_T(A,x); \text{data}_T\rangle\rangle$. By definition, $\mathcal{O}(x)$ and data_T are identical on datasteps(x), so it suffices to show @visible$_L$[x](A,x) = $\text{visible}_T(A,x)$.

@visible$_L$[x](A,x) $\subseteq \text{visible}_T(A,x)$ by Lemma 7.3.2h. So take $B \in \text{visible}_T(A,x)$,
$\Rightarrow$ $B \in$ @datasteps$_L$[x](A,x) by Lemma 7.3.2e,
$\Rightarrow$ $B \in$ @visible$_L$[x](A,x) $\vee$ $B \in$ @dead$_L$[x](A,x) by P3.4b.

But $B \in$ @dead$_L$[x](A,x) $\Rightarrow$ $B \in \text{dead}_T(A,x)$ (by Lemma 7.3.2i) -- a contradiction;
$\Rightarrow$ $B \in$ @visible$_L$[x](A,x).

d. $B \in \text{visible}_T(A,x) \Rightarrow B \in @\text{visible}_L[x](A,x)$ (as in (c) above),
$\Rightarrow \text{anc}(A) \cap \text{ABORTS}_T(A{\downarrow}B) = \emptyset$ by P3.4d. ∎

Lemma 7.4.4: h is a possibilities map relative to I3.

Proof: Follows immediately from Lemmas 7.4.1, 7.4.2, 7.4.3, and from Lemma 4.2.4.2.4. ∎

Theorem 7.4.5: h is a possibilities map, and I3 is invariant in L3.

Proof: By Lemma 7.3.2, I3 is invariant relative to h. By Lemma 7.4.4, h is a possibilities map relative to I3. We apply Lemma 4.2.4.2.6 to conclude that h is a possibilities map, and I3 is an invariant. ∎

Since h_{32} is a possibilities map which fixes T, all invariants and pair-invariants from L2 carry down to L3. Let J3 denote the conjunction of all pair-properties from Lemma 7.3.1. We summarize the invariants for L3 as follows:

Lemma 7.4.6: I3 is invariant in L3, Ia is invariant in L3, J3 is pair-invariant in L3, and Ja is pair-invariant in L3.

Proof: Invariance of I3 is shown in Theorem 7.4.5. J3 is pair-invariant in L3 by Lemma 7.3.1. Since h_{32} is a possibilities map which fixes T, and Ia is invariant for T in L2 (by Lemma 6.5.2), Ia is invariant for T in L3, by Lemma 4.2.4.3.5. Similarly since Ja is pair-invariant for T in L2 (by Lemma 6.5.2), Ja is pair-invariant for T in L3, by Lemma 4.2.4.3.5. ∎

8. Value Maps -- A Model of Atomic Objects

At Level 4 we introduce *value maps* as a data structure for keeping lock and version information about objects. A value map is a mapping from each object to a "stack of versions" for that object; each version is associated with an action that holds a lock on that object. This data structure corresponds roughly to the implementation of atomic objects as described in [Moss81]. In Moss's locking scheme, a lock can be held on an atomic object at each level in the action tree. This scheme constrains all holders of a lock on a particular object to be *related.* We note again that we are dealing only with *mutual exclusion* locks. Moss develops a more general locking protocol which distinguishes between read locks and write locks.

We regard these value maps as an abstraction of the information which is already present in the local UAS's at each object. In this sense the value maps introduce no new information into the state. As we stressed in Chapter 4, the state in an event-state algebra is simply one convenient way of capturing execution histories. Value maps are a convenient abstraction of execution histories because the preconditions on a perform event can be stated easily in terms of value maps.

Level 4 is no more "localized" than Level 3. The events in Level 4 are identical to those in Level 3 (though transitions and preconditions are reformulated in terms of value maps), and the event mapping h_{43} is the identity. In particular, then, the communications events at Level 4 are still very simple, and they do not include enough information to allow localization of the orphan detection precondition. The non-local orphan detection precondition appears unchanged at Level 4.

8.1 Level 4 Algebra

$L4 = (\mathcal{E}_4, \Sigma_4, \sigma_4, \tau_4)$

<u>State Space:</u>

$\Sigma_4 = \{\langle T,L,V \rangle\}$, where the components are:

T - the global state, an augmented action tree (as in L2),

L - the local state (mapping loc to UAS) (as in L3),

V - a value map.

A *value map* gives a set of values for each object -- one value for each action which "holds a lock" (and thus a version) on that object.

V: obj $\times$ act $\rightarrow$ values $\cup \{\perp\}$

(where $\forall A \in$ act, $x \in$ obj, $V(x,A) \in$ values$(x) \cup \{\perp\}$).

Define $V(x) = \{A \in \text{act}: V(x,A) \neq \perp\}$. ($V(x)$ represents the actions which hold locks on object x.)

If $V(x)$ forms an ancestor chain, then define

$V(x)$.holder = the lowest (in anc-order) element of ($V(x)$), i.e.,

$V(x)$.holder $\in V(x)$, and $\forall B \in V(x)$, $V(x)$.holder $\in$ desc(B).

(If $V(x)$ does not form an ancestor chain, then $V(x)$.holder is undefined. For reachable states in L4, $V(x)$ will always form an ancestor chain (see below).)

If $V(x)$.holder is defined, then define $V(x)$.value = $V(x,V(x)$.holder). $V(x)$.value denotes the "current" value of object x which will be seen by any datastep accessing x.

Initial State:

$\sigma_4 = \langle T_0, L_0, V_0 \rangle$, where

$\forall x \in$ obj, $V_0(x,U) =$ init(x),

$V_0(x,A) = \perp$, $\forall A \neq U$.

Events:

$\mathcal{E}_4 = \mathcal{E}_3$ (The sets of events are identical in Levels 3 and 4, although preconditions and transitions differ.)

Transition Relation

Let $e \in \mathcal{E}_4$, $\langle T,L,V \rangle \in \Sigma_4$, $\langle T,L,V \rangle e = \langle T1,L1,V1 \rangle$.

1. create A (A $\in$ act - {U})

PRECONDITIONS:

a. $A \notin @vertices_L[creator(A)]$

b. $parent(A) \in @active_L[creator(A)]$

c. $(B,A) \in seq, B \neq A \implies B \in @done_L[creator(A)]$

TRANSITIONS:

a. $vertices_{T1} \leftarrow vertices_T \cup \{A\}$

b. $status_{T1}(A) \leftarrow$ 'active'

c. $@vertices_{L1}[creator(A)] \leftarrow @vertices_L[creator(A)] \cup \{A\}$

d. $@status_{L1}[creator(A)](A) \leftarrow$ 'active'

2. <u>commit A</u> $(A \in act - \{U\} - accesses)$

PRECONDITIONS:

a. $A \in @active_L[A]$

b. $@children_L[A](A) \subseteq @done_L[A]$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'committed'

b. $@status_{L1}[A](A) \leftarrow$ 'committed'

3. <u>abort A</u> $(A \in act - \{U\})$

PRECONDITIONS:

a. $A \in @active_L[A]$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'aborted'

b. $@status_{L1}[A](A) \leftarrow$ 'aborted'

4. <u>perform A.u</u> $(A \in accesses(x), u \in values(x))$

PRECONDITIONS:

a. $A \in @active_L[x]$

b. $A \in$ prop-desc(V(x).holder)

c. $u = V(x).value$

d. $anc(A) \cap @aborted_L[x] = \emptyset$

e. $B \in @visible_L[x](A,x) \Rightarrow anc(A) \cap ABORTS_T(A{\downarrow}B) = \emptyset$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'committed'

b. $@status_{L1}[x](A) \leftarrow$ 'committed'

c. $label_{T1}(A) \leftarrow u$

d. $data_{T1} \leftarrow data_T \cup \{(B,A): B \in datasteps_T(x)\} \cup \{(A,A)\}$

e. $V1(x,parent(A)) \leftarrow update(A)(u)$

5. <u>@create[α] A</u> $(A \in act - \{U\}, \alpha \in loc)$

PRECONDITIONS:

a. $A \in @vertices_L[\beta]$, for some $\beta \in loc$

TRANSITIONS:

a. $@vertices_{L1}[\alpha] \leftarrow @vertices_L[\alpha] \cup \{A\}$

b. $A \notin @vertices_L[\alpha] \Rightarrow @status_{L1}[\alpha](A) \leftarrow$ 'active'

6. <u>@commit[α] A</u> $(A \in act - \{U\}, \alpha \in loc)$

PRECONDITIONS:

a. $A \in @committed_L[\beta]$, for some $\beta \in loc$

TRANSITIONS:

a. $@vertices_{L1}[\alpha] \leftarrow @vertices_L[\alpha] \cup \{A\}$

b. $@status_{L1}[\alpha](A) \leftarrow$ 'committed'

c. $\alpha \in obj,\ V(\alpha,A) \neq \perp \ \Rightarrow$
$V1(\alpha,A) \leftarrow \perp$
$V1(\alpha,parent(A)) \leftarrow V(\alpha,A)$

7. <u>$@abort[\alpha]$ A</u> $(A \in act - \{U\},\ \alpha \in loc)$

PRECONDITIONS:

a. $A \in @aborted_L[\beta]$, for some $\beta \in loc$

TRANSITIONS:

a. $@vertices_{L1}[\alpha] \leftarrow @vertices_L[\alpha] \cup \{A\}$

b. $@status_{L1}[\alpha](A) \leftarrow$ 'aborted'

c. $\alpha \in obj,\ B \in desc(A) \ \Rightarrow$
$V1(\alpha,B) \leftarrow \perp$

Local create, commit, and abort events are identical in L4 and L3. The preconditions on perform events are given in terms of the value map. Note that we include a "local orphan detection" precondition (P4.4d): this condition is necessary for the value map to hold the proper versions, but it is not *sufficient* to detect all harmful orphans. Thus we retain the non-local orphan detection precondition (P4.4e).

The effect of a perform event is to update the "current" version. A lock on the current version is held by the *parent* of the datastep immediately after the perform event. The value map is updated by commit and abort messages: a commit message for an action releases a lock held by that action to its parent (and the parent inherits its child's version). An abort message for an action releases all locks held by descendants of that action (and the versions are discarded).

8.2 Specification of Mapping h_{43}

We define a (single-state) mapping from L4 to L3, h_{43}: L4 → L3. (We abbreviate "h_{43}" as "h" in this chapter.)

<u>State Mapping</u>

h: $\Sigma_4 \rightarrow \Sigma_3$ is defined by $h(\langle T,L,V\rangle) = \langle T,L\rangle \ \forall \langle T,L,V\rangle \in \Sigma_4$. Thus h fixes $\langle T,L\rangle$.

<u>Event Mapping</u>

h: $\mathcal{E}_4 \rightarrow \mathcal{E}_3^*$ is the identity mapping on events, i.e. $h(e) = e \quad \forall e \in \mathcal{E}_4$.

8.3 Level 4 Invariants

The following invariants relate the information in the value map, V, to the local data structure, L. We show these invariants relative to mapping h: Since h fixes $\langle T,L\rangle$, all invariants and pair-invariants for $\langle T,L\rangle$ in L3 can be applied to the proofs (by Lemma 4.2.4.3.6). (Recall that we have shown in Lemma 7.4.6 that I3 and Ia are invariant in L3, and J3 and Ja are pair-invariant in L3.)

Lemma 8.3.1: Let $\langle T,L,V\rangle \in \mathcal{R}_4$. Then the following are invariant relative to h:

$(\forall x \in \text{obj})$ (let $M = L(x)$, and let $\mathcal{O} = \text{order}(T)$):

a. $B \in V(x) \Rightarrow B$ is live in M

b. V(x) forms an ancestor chain

c. $V(x) \cap \text{accesses} = \emptyset$

d. $B \in \text{datasteps}_M(x) \Rightarrow$
B is dead in M $\vee$ $\exists B' \in \text{anc}(B) \cap V(x)$: $B \in \text{visible}_M(B')$

e. $B \in V(x)$, $\text{v-prop-desc}_M(B) \cap V(x) = \emptyset \Rightarrow$
$V(x,B) = \text{result}(x,s)$, where $s = \langle\langle \text{visible}_M(B,x); \mathcal{O}(x)\rangle\rangle$

f. $H = V(x).\text{holder}$, $B \in \text{desc}(H)$, B is live in M
$\Rightarrow \text{visible}_M(B,x) = \text{visible}_M(H,x)$

g. $B \in V(x), C \in \text{datasteps}_M \Rightarrow$ exactly one of following holds:

1. $C \in \text{visible}_M(B)$

2. $C \in \text{dead}_M(B)$

3. $\exists C' \in \text{prop-desc}(B) \cap \text{prop-anc}(C) \cap V(x)$:
   $(C \in \text{visible}_M(C')) \wedge (C' \notin \text{visible}_M(B))$

Proof: We show below that (f) and (g) follow from (a) - (e). It is trivial to show that (a) - (e) are 0-invariant (i.e. that they hold for σ_4). The proofs that (b) and (c) are invariant relative to h are straightforward; we will argue (a), (d), and (e).

For the induction step, let $\langle T,L,V\rangle \in \mathfrak{R}_4 \cap \text{PRE}_5(e)$, and assume that (a) - (g) hold for $\langle T,L,V\rangle$. Let $\langle T',L',V'\rangle = \langle T,L,V\rangle e$. Let $\mathcal{O} = \text{order}(T)$, $\mathcal{O}' = \text{order}(T')$, $M = L(x)$, $M' = L'(x)$. We must show that (a), (d), and (e) hold for $\langle T',L',V'\rangle$. By Lemma 4.2.4.3.6, we can assume that $\langle T,L,V\rangle$ and $\langle T',L',V'\rangle$ satisfy any invariants from 13 or Ia, and we can assume that $(\langle T,L,V\rangle,\langle T',L',V'\rangle)$ satisfy any pair-invariants from J3 or Ja.

Since properties (a), (d), and (e) depend only on $V(x)$, committed_M, aborted_M, and $\mathcal{O}(x)$, we need only consider events, e, which modify these components. By inspection, these events are $\{\text{abort } A, \text{perform } A,u: A \in \text{accesses}(x)\} \cup \{\text{@commit}[x]\ A, \text{@abort}[x]\ A\}$.

1. abort A, $A \in \text{accesses}(x)$

   $\text{aborted}_{M'} = \text{aborted}_M \cup \{A\}$.
   $\text{committed}_{M'} = \text{committed}_M$.
   $V' = V, \mathcal{O}' = \mathcal{O}$.

   a. $B \in V'(x) \Rightarrow B \in V(x), \Rightarrow B$ is live in M (by (a)).
   But by (c), $V(x) \cap \text{accesses} = \emptyset$,
   $\Rightarrow B \notin \text{accesses}, \Rightarrow \text{anc}(B) \cap \{A\} = \emptyset, \Rightarrow B$ is live in M'.

   d. $B \in \text{datasteps}_{M'}(x) \Rightarrow B \in \text{datasteps}_M(x)$,
   $\Rightarrow B$ is dead in M $\vee\ \exists B' \in \text{anc}(B) \cap V(x): B \in \text{visible}_M(B')$. But
   B is dead in M $\Rightarrow$ B is dead in M' by 7.3.1b.
   $B' \in \text{anc}(B) \cap V(x) \Rightarrow B' \in \text{anc}(B) \cap V'(x)$, and
   $B \in \text{visible}_M(B') \Rightarrow B \in \text{visible}_{M'}(B')$, by 7.3.1c.

e. Immediate since all components unchanged.

2. perform A,u, $A \in accesses(x)$

$\mathcal{O}'(x) = \mathcal{O}(x) \cup \{(B,A): B \in datasteps_M(x)\} \cup \{(A,A)\}$.
$V'(x,parent(A)) = update(A)(V(x).value)$.
$V'(x,B) = V(x,B)\ \ \forall B \neq parent(A)$.
$V'(x) = V(x) \cup \{parent(A)\}$.
$committed_{M'} = committed_M \cup \{A\}$.
$aborted_{M'} = aborted_M$ (thus live in M $\Rightarrow$ live in M').

Note that $A \in prop\text{-}desc(V(x).holder)$, by P4.4b, and A is live in M, by P4.4d.

a. $B \in V'(x) \Rightarrow B \in V(x) \vee B = parent(A)$.
If $B \in V(x)$, then B is live in M, so B is live in M'.
If $B = parent(A)$, then $anc(B) \subseteq anc(A)$, and A is live in M. Thus B is live in M, and B is live in M'.

d. $B \in datasteps_{M'}(x) \Rightarrow B \in datasteps_M(x) \vee B = A$
If $B \in datasteps_M(x)$, then B is dead in M $\vee\ \exists B' \in anc(B) \cap V(x)$: $B \in visible_M(B')$.

If B is dead in M, then B is dead in M' by 7.3.1b.
If $B' \in anc(B) \cap V(x)$, then $B' \in anc(B) \cap V'(x)$, and $B \in visible_M(B') \Rightarrow B \in visible_{M'}(B')$ by 7.3.1c.

If $B = A$, then take $B' = parent(A)$, because $parent(A) \in anc(A) \cap V'(x)$, and $A \in visible_{M'}(parent(A))$.

e. $B \in V'(x)$, $v\text{-}prop\text{-}desc_{M'}(B) \cap V'(x) = \emptyset$.
$B \in V'(x) \Rightarrow B \in V(x) \vee B = parent(A)$.

Case 1: $B \in V(x)$.
$v\text{-}prop\text{-}desc_M(B) \subseteq v\text{-}prop\text{-}desc_{M'}(B)$, and $V(x) \subseteq V'(x)$
$\Rightarrow v\text{-}prop\text{-}desc_M(B) \cap V(x) = \emptyset$.
Thus $V(x,B) = V'(x,B) = result(x,s)$, where $s = \langle\langle visible_M(B,x); \mathcal{O}(x)\rangle\rangle$.

We must show $V'(x,B) = result(x,s')$, where $s' = \langle\langle visible_{M'}(B,x); \mathcal{O}'(x)\rangle\rangle$. Since $\mathcal{O}(x) \subseteq \mathcal{O}'(x)$, it suffices to show $visible_{M'}(B,x) = visible_M(B,x)$. Since A is the only action whose status changes from M to M', $visible_{M'}(B,x) = visible_M(B,x)$ unless $A \in visible_{M'}(B,x)$. So assume $A \in visible_{M'}(B,x)$.

Since $parent(A) \in desc(V(x).holder)$ (by P4.4b), $parent(A) \in prop\text{-}desc(B)$ (since we assumed $B \neq parent(A)$). But $A \in$

$\text{visible}_{M'}(B,x) \Rightarrow \text{parent}(A) \in \text{v-prop-desc}_{M'}(B) \cap V'(x)$ -- a contradiction.

Case 2: $B = \text{parent}(A)$.
If $B = \text{parent}(A)$, then $V'(x,B) = \text{update}(A)(V(x).\text{value})$.
Let $H = V(x).\text{holder}$. Then by definition of holder $\text{v-prop-desc}_M(H) \cap V(x) = \emptyset$. Thus $V(x,H) = \text{result}(x,s)$, where $s = \langle\langle \text{visible}_M(H,x); O(x)\rangle\rangle$, $\Rightarrow V'(x,B) = \text{result}(x,s')$, where $s' = \langle\langle \text{visible}_M(H,x) \cup \{A\}; O'(x)\rangle\rangle$.

We must show that $\text{visible}_M(H,x) \cup \{A\} = \text{visible}_{M'}(\text{parent}(A),x)$. Clearly $\text{visible}_{M'}(\text{parent}(A),x) = \text{visible}_M(\text{parent}(A),x) \cup \{A\}$, so we show $\text{visible}_M(H,x) = \text{visible}_M(\text{parent}(A),x)$.

But $A \in \text{prop-desc}(H) \Rightarrow \text{parent}(A) \in \text{desc}(H)$, and A live in $M \Rightarrow \text{parent}(A)$ live in M. Thus $\text{visible}_M(H,x) = \text{visible}_M(\text{parent}(A),x)$, by (f).

3. @commit[x] A

There are two cases:

(1) If $V(x,A) \neq \perp$, then
$V'(x) = V(x) - \{A\} \cup \{\text{parent}(A)\}$,
$V'(x,A) = \perp$,
$V'(x,\text{parent}(A)) = V(x,A)$.
(2) If $V(x,A) = \perp$, then $V(x) = V'(x)$.

$\text{committed}_{M'} = \text{committed}_M \cup \{A\}$.
$\text{aborted}_{M'} = \text{aborted}_M$ (thus live in $M \Rightarrow$ live in M').

$\text{datasteps}_{M'}(x) = \text{datasteps}_M(x)$, since $A \in \text{accesses}(x) \Rightarrow A \in \text{@committed}_L[\beta]$, for some β, by P4.6a, $\Rightarrow A \in \text{committed}_T$ by Lemma 7.3.2f, $\Rightarrow A \in \text{@committed}_M$, by Lemma 7.3.2b.

a. For case (1), $B \in V'(x) \Rightarrow B \in V(x) \Rightarrow B$ is live in $M \Rightarrow B$ is live in M'.
For case (2), $B \in V'(x) \Rightarrow B \in V(x)$ or $B = \text{parent}(A)$. If $B \in V(x)$ then the proof is identical to case (1), otherwise we know $A \in V(x)$, and A is live in M. It follows that B is live in $M \Rightarrow B$ is live in M'.

d. $B \in \text{datasteps}_{M'}(x) \Rightarrow B \in \text{datasteps}_M(x)$
$\Rightarrow B$ is dead in $M \vee \exists B' \in \text{anc}(B) \cap V(x)$: $B \in \text{visible}_M(B')$.

If B is dead in M then B is dead in M', so suppose that $B' \in \text{anc}(B) \cap V(x)$.

For case (1), $V'(x) = V(x)$, $\Rightarrow$ $B' \in \text{anc}(B) \cap V'(x)$, and $B \in \text{visible}_M(B') \Rightarrow B \in \text{visible}_{M'}(B')$.

For case (2), $A \in V(x)$, and $V'(x) = V(x) - \{A\} \cup \{\text{parent}(A)\}$.
If $B' \neq A$, then $B' \in \text{anc}(B) \cap V'(x)$ and $B \in \text{visible}_{M'}(B')$ as above.

If $B' = A$, then $B \in \text{visible}_M(A)$, and $A \in \text{visible}_{M'}(\text{parent}(A))$ (since $A \in \text{committed}_{M'}$.
Thus $B \in \text{visible}_{M'}(\text{parent}(A))$, and $\text{parent}(A) \in \text{anc}(B) \cap V'(x)$.

e. $B \in V'(x)$, $\text{v-prop-desc}_{M'}(B) \cap V'(x) = \emptyset$

Case 1: $A \in V(x)$, $\Rightarrow$ $V'(x) = V(x) - \{A\} \cup \{\text{parent}(A)\}$. Thus $B \neq A$.

Case 1a: $B \neq \text{parent}(A)$.
$\Rightarrow$ $B \in V(x)$. But $\text{v-prop-desc}_M(B) \subseteq \text{v-prop-desc}_{M'}(B)$, and $V(x) - \{A\} \subseteq V'(x)$. Thus $(V(x) - \{A\}) \cap \text{v-prop-desc}_M(B) = \emptyset$.

But if $A \in \text{v-prop-desc}_M(B)$ and $B \neq \text{parent}(A)$, then $\text{parent}(A) \in \text{v-prop-desc}_M(B) \cap V'(x)$ -- a contradiction. Thus $\text{v-prop-desc}_M(B) \cap V(x) = \emptyset$.

Thus $V(x,B) = V'(x,B) = \text{result}(x,s)$, where $s = \langle\langle \text{visible}_M(B,x); O(x) \rangle\rangle$.
We show that $\text{visible}_M(B,x) = \text{visible}_{M'}(B,x)$. Clearly $\text{visible}_M(B,x) \subseteq \text{visible}_{M'}(B,x)$. Let $D \in \text{visible}_{M'}(B,x) - \text{visible}_M(B,x)$; we show that the existence of D leads to a contradiction.

We apply (g) to D and B: We cannot have $D \in \text{visible}_M(B)$ by our assumption. If $D \in \text{dead}_M(B)$, then $D \notin \text{visible}_{M'}(B)$ -- a contradiction. Thus we are left with the third case: $\exists D' \in \text{prop-desc}(B) \cap \text{prop-anc}(D) \cap V(x)$: $(D \in \text{visible}_M(D')) \wedge (D' \notin \text{visible}_M(B)))$.

$D \in \text{visible}_{M'}(B) \Rightarrow D' \in \text{v-prop-desc}_{M'}(B)$. But if $D' \neq A$, then $D' \in V'(x) \cap \text{v-prop-desc}_{M'}(B)$ -- a contradiction. If $D' = A$, then $\text{parent}(A) \in V'(x) \cap \text{v-prop-desc}_{M'}(B)$ -- a contradiction.

Case 1b: $B = \text{parent}(A)$.
$\text{v-prop-desc}_M(A) \subseteq \text{v-prop-desc}_{M'}(A) \subseteq \text{v-prop-desc}_{M'}(\text{parent}(A))$, since $A \in \text{visible}_{M'}(\text{parent}(A))$, and $V(x) \subseteq V'(x) \cup \{A\}$. Thus $\text{v-prop-desc}_M(A) \cap V(x) = \emptyset$.

Thus $V(x,A) = V'(x,\text{parent}(A)) = \text{result}(x,s)$, where $s = \langle\langle \text{visible}_M(A,x); O(x) \rangle\rangle$. But $\text{visible}_{M'}(\text{parent}(A),x) = \text{visible}_M(A,x)$ (since $A \in \text{committed}_{M'}$), and $O'(x) = O(x)$.

Case 2: A $\notin$ V(x), $\Rightarrow$ V'(x) = V(x). Thus B $\in$ V(x). v-prop-desc$_{M}$(B) $\subseteq$ v-prop-desc$_{M'}$(B), $\Rightarrow$ v-prop-desc$_{M}$(B) $\cap$ V(x) = $\emptyset$. Thus V(x,B) = V'(x,B) = result(x,s), where s = <<visible$_{M}$(B,x); $\mathcal{O}$(x)>>. We must show visible$_{M}$(B,x) = visible$_{M'}$(B,x). Clearly visible$_{M}$(B,x) $\subseteq$ visible$_{M'}$(B,x). Let D $\in$ visible$_{M'}$(B,x) - visible$_{M}$(B,x); we show that the existence of D leads to a contradiction.

As in case (1a), we apply (g) to D and B; the only possible case is the third: $\exists$D' $\in$ prop-desc(B) $\cap$ prop-anc(D) $\cap$ V(x): (D $\in$ visible$_{M}$(D')) $\wedge$ (D' $\notin$ visible$_{M}$(B))).

But D $\in$ visible$_{M'}$(B) $\Rightarrow$ D' $\in$ visible$_{M'}$(B) $\cap$ V'(x) -- a contradiction.

4. @abort[x] A

V'(x) = V(x) - desc(A).
B $\in$ V'(x) $\Rightarrow$ V'(x,B) = V(x,B).
committed$_{M'}$ = committed$_{M}$.
aborted$_{M'}$ = aborted$_{M}$ $\cup$ {A}.
$\mathcal{O}$'(x) = $\mathcal{O}$(x).

a. B $\in$ V'(x) $\Rightarrow$ B $\in$ V(x), B $\notin$ desc(A). Thus anc(B) $\cap$ aborted$_{M}$ = $\emptyset$, $\Rightarrow$ anc(B) $\cap$ aborted$_{M'}$ = $\emptyset$, since B $\notin$ desc(A) and aborted$_{M'}$ = aborted$_{M}$ $\cup$ {A}. Thus B is live in M'.

d. B $\in$ datasteps$_{M'}$(x) $\Rightarrow$ B $\in$ datasteps$_{M}$(x) $\Rightarrow$ B is dead in M $\vee$ $\exists$B' $\in$ anc(B) $\cap$ V(x): B $\in$ visible$_{M}$(B').

If B is dead in M, then B is dead in M'.

If B' $\in$ anc(B) $\cap$ V(x), and B' $\notin$ desc(A), then B' $\in$ anc(B) $\cap$ V'(x), and B $\in$ visible$_{M'}$(B') since B $\in$ visible$_{M}$(B').

If B' $\in$ desc(A) then B $\in$ desc(A), $\Rightarrow$ A $\in$ anc(B) $\cap$ aborted$_{M'}$ $\Rightarrow$ B is dead in M'.

e. B $\in$ V'(x), v-prop-desc$_{M'}$(B) $\cap$ V'(x) = $\emptyset$,
$\Rightarrow$ B $\in$ V(x), and B $\notin$ desc(A).

Suppose C $\in$ v-prop-desc$_{M}$(B) $\cap$ V(x); then C $\in$ v-prop-desc$_{M'}$(B) $\cap$ V(x), $\Rightarrow$ C $\in$ desc(A) (since V'(x) = V(x) - desc(A)).
But B $\notin$ desc(A), so A $\in$ prop-desc(B) $\cap$ anc(C). Then C $\notin$ visible$_{M}$(B) -- a contradiction.

Thus v-prop-desc$_M$(B) $\cap$ V(x) = $\emptyset$, $\Rightarrow$ V(x,B) = V'(x,B) = result(x,s), where s = $\langle\langle$visible$_M$(B,x); O(x)$\rangle\rangle$ = $\langle\langle$visible$_{M'}$(B,x); O'(x)$\rangle\rangle$.

Proof of (g): First we show that at least one of the three conditions must hold:

B $\in$ V(x), C $\in$ datasteps$_M$. But by (d), either C is dead in M, or $\exists$C' $\in$ anc(C) $\cap$ V(x): C $\in$ visible$_M$(C').

If C is dead in M, then either C $\in$ dead$_M$(B), or lca(B,C) is dead in M. But if lca(B,C) is dead in M, then B is dead in M, which contradicts (a). Thus we have case (g2).

So suppose $\exists$C' $\in$ anc(C) $\cap$ V(x): C $\in$ visible$_M$(C'). If C' $\in$ visible$_M$(B), then C $\in$ visible$_M$(B), which is case (g1). If C' $\notin$ visible$_M$(B), then (B,C') $\in$ related, since V(x) forms an ancestor chain (by (b)). But if C' $\in$ anc(B), then C' $\in$ visible$_M$(B). Thus C' $\in$ prop-desc(B) $\cap$ prop-anc(C) $\cap$ V(x), which is case (g3).

To see that only one condition can hold, it is clear that (g1) and (g2) are mutually exclusive, and that (g1) and (g3) are mutually exclusive. If (g3) holds, then C $\in$ visible$_M$(C'), and C' $\in$ V(x). But by (a), C' must be live in M, so C must be live in M; thus C $\notin$ dead$_M$(B). Thus (g2) and (g3) are mutually exclusive.

Proof of (f): H = V(x).holder, B $\in$ desc(H), B is live in M. We show visible$_M$(B,x) = visible$_M$(H,x). Since B $\in$ desc(H), it is obvious that visible$_M$(H,x) $\subseteq$ visible$_M$(B,x). If B = H then the result is obvious, so assume B $\in$ prop-desc(H). Suppose D $\in$ visible$_M$(B,x); we show D $\in$ visible$_M$(H,x). Let L = lca(B,D).

D $\in$ visible$_M$(B,x) $\Rightarrow$ D $\in$ visible$_M$(L), $\Rightarrow$ L $\in$ prop-desc(H), since D $\notin$ visible$_M$(H).

Now we apply (g) to D and H: If (g2) holds, then D is dead to H. But since B is live in M, L is not dead to H; thus D must be dead to B. But D $\in$ dead$_M$(B) contradicts D $\in$ visible$_M$(B).

(g3) cannot hold, because by definition of holder there is no D' $\in$ prop-desc(H) $\cap$ V(x).

Thus (g1) must hold, $\Rightarrow$ D $\in$ visible$_M$(H). ∎

8.4 Proof of Possibilities Map for h_{43}

We now show that h is a possibilities map. Let I4 be the conjunction of all properties in Lemma 8.3.1. We will show that h is a possibilities map relative to I4.

Lemma 8.4.1: h preserves initial states.

Proof: Immediate, since $h(\langle T_0,L_0,V_0\rangle) = \langle T_0,L_0\rangle$. ∎

Lemma 8.4.2: h preserves transitions relative to I4.

Proof: We must show that if $\langle T,L,V\rangle \in PRE_4(e) \cap \mathfrak{R}_4 \cap I4$, and $h(\langle T,L,V\rangle) \in PRE_3(h(e)) \cap \mathfrak{R}_3$, then $h(\langle T,L,V\rangle e) = h(\langle T,L,V\rangle)h(e)$.

But $h(\langle T,L,V\rangle) = \langle T,L\rangle$ and $h(e) = e$, so we must show the following:

If $\langle T,L,V\rangle \in PRE_4(e) \cap \mathfrak{R}_4 \cap I4$, and $\langle T,L\rangle \in PRE_3(h(e)) \cap \mathfrak{R}_2$, and $\langle T,L,V\rangle e = \langle T1,L1,V1\rangle$, then $\langle T1,L1\rangle = \langle T,L\rangle e$.

It is easily verified by inspection that the effects of all events on T and L are identical in L3 and L4; thus h preserves transitions relative to I4. ∎

Lemma 8.4.3: h preserves preconditions relative to I4.

Proof: We must show that if $\langle T,L,V\rangle \in PRE_4(e) \cap \mathfrak{R}_4 \cap I4$, and $h(\langle T,L,V\rangle) \in \mathfrak{R}_3$, then $h(\langle T,L,V\rangle) \in PRE_3(h(e))$.

Since $h(\langle T,L,V\rangle) = \langle T,L\rangle$, and $h(e) = e$, we must show
$\langle T,L,V\rangle \in PRE_4(e) \cap \mathfrak{R}_4 \cap I4, \langle T,L\rangle \in \mathfrak{R}_3 \Rightarrow \langle T,L\rangle \in PRE_3(e)$.

Preservation of preconditions is easily verified by inspection for all events other than perform, since preconditions are identical in L3 and L4.

We prove preservation of preconditions for event e = perform A,u:

a. P4.4a $\Rightarrow$ A $\in$ @active$_L$[x].

b. B $\in$ @datasteps$_L$[x](x) $\Rightarrow$ B is dead in L(x) $\vee$ $\exists$B' $\in$ anc(B) $\cap$ V(x): B $\in$ @visible$_L$[x](B'), by 8.3.1d.

If B is dead in L(x), then anc(B) $\cap$ @aborted$_L$[x] $\neq \emptyset$. But P4.4d $\Rightarrow$ anc(A) $\cap$ @aborted$_L$[x] = $\emptyset$, $\Rightarrow$ anc(lca(A,B)) $\cap$ @aborted$_L$[x] = $\emptyset$.

Thus anc(B) $\cap$ prop-desc(lca(A,B)) $\cap$ @aborted$_L$[x] $\neq \emptyset$, $\Rightarrow$ B $\in$ @dead$_L$[x](A).

If B' $\in$ anc(B) $\cap$ V(x), then B' $\in$ anc(V(x).holder).
But A $\in$ prop-desc(V(x).holder) by P4.4b, $\Rightarrow$ A $\in$ prop-desc(B').
Thus B $\in$ @visible$_L$[x](B') $\Rightarrow$ B $\in$ @visible$_L$[x](A), by Lemma 2.2.3.1d.

c. P4.4c $\Rightarrow$ u = V(x).value. Let H = V(x).holder (then u = V(x,H)).

By 8.3.1e, V(x,H) = result(x,s), where s = <<@visible$_L$[x](H,x); O(x)>>.

But A $\in$ prop-desc(H) by P4.4b, and A is live in L(x) by P4.4d,
$\Rightarrow$ @visible$_L$[x](H,x) = @visible$_L$[x](A,x), by 8.3.1f.

Thus u = result(x,s'), where s' = <<@visible$_L$[x](A,x); O(x)>>.

d. B $\in$ @visible$_L$[x](A,x) $\Rightarrow$ anc(A) $\cap$ ABORTS$_T$(A$\downarrow$B) = $\emptyset$, directly by P4.4e.
∎

Lemma 8.4.4: h is a possibilities map relative to I4.

Proof: Follows immediately from Lemmas 8.4.1, 8.4.2, 8.4.3, and from Lemma 4.2.4.2.4. ∎

Theorem 8.4.5: h is a possibilities map, and I4 is invariant in L4.

Proof: By Lemma 8.3.1, I4 is invariant relative to h. By Lemma 8.4.4, h is a possibilities map relative to I4. We apply Lemma 4.2.4.2.6 to conclude that h is a possibilities map, and I4 is an invariant. ∎

Since h_{43} is a possibilities map which fixes <T,L>, all invariants and pair-invariants from L3 carry down to L4. In the following Lemma we summarize all the known invariants and pair-invariants for L4:

Lemma 8.4.6: Ia, I3, and I4 are invariant in L4, and Ja, J3 are pair-invariant in L4.

Proof: Invariance of I4 is shown in Theorem 8.4.5. Since h_{43} is a possibilities map which fixes <T,L>, and Ia, I3 are invariant in L3, Ia and I3 are invariant in L4, by Lemma 4.2.4.3.5. Similarly since Ja, J3 are pair-invariant in L3, Ja and J3 are pair-invariant in L4, by Lemma 4.2.4.3.5. ▮

9. Fully Localized Models

At Level 5 we completely localize all preconditions by "piggybacking" abort information on communications steps. This additional information flow allows us to replace the orphan detection precondition (P4.4c) with a local check for orphans. Other than the new abort information in communication steps (and the elimination the non-local orphan precondition), Level 5 is identical to Level 4.

Because all preconditions are localized at Level 5, we can project out the "virtual" global state to define Level 6.

9.1 Level 5 Algebra

$L5 = (\mathcal{E}_5, \Sigma_5, \sigma_5, \tau_5)$

State Space:

$\Sigma_5 = \Sigma_4 = \{\langle T,L,V\rangle\}$, where the components are:

T - the global state, an augmented action tree (as in L2),
L - local UAS's (as in L3),
V - value maps (as in L4).

Initial State:

$\sigma_5 = \sigma_4 = \langle T_0, L_0, V_0\rangle$.

Events:

The local steps are identical in L5 and L4, but for the communications events we introduce an explicit "sender" of information. (Thus communications events are now parameterized by two locations: the sender and the receiver.) This modification is necessary to describe precisely the set of aborts which must be piggybacked on a communications event. (In fact this set will be all aborts known to the sender).

Communications events:

@create[β,α] A,d -- send create message from β to α with aborts d

@commit[β,α] A,d -- send commit message from β to α with aborts d

@abort[β,α] A -- send abort message from β to α

The parameter "d" of create and commit messages models the DONE lists in remote invocation and commit messages.

As in previous levels, the transition relation will be defined so that each communications event is idempotent.

Transition Relation

Let e $\in \mathcal{E}_5$, <T,L,V> $\in \Sigma_5$, <T,L,V>e = <T1,L1,V1>.

1. create A (A $\in$ act - {U})

PRECONDITIONS:

a. A $\notin$ @vertices$_L$[creator(A)]

b. parent(A) $\in$ @active$_L$[creator(A)]

c. (B,A) $\in$ seq, B $\neq$ A $\Rightarrow$ B $\in$ @done$_L$[creator(A)]

TRANSITIONS:

a. vertices$_{T1}$ $\leftarrow$ vertices$_T$ $\cup$ {A}

b. status$_{T1}$(A) $\leftarrow$ 'active'

c. @vertices$_{L1}$[creator(A)] $\leftarrow$ @vertices$_L$[creator(A)] $\cup$ {A}

d. @status$_{L1}$[creator(A)](A) $\leftarrow$ 'active'

2. commit A (A $\in$ act - {U} - accesses)

PRECONDITIONS:

a. A $\in$ @active$_L$[A]

b. $@children_L[A](A) \subseteq @done_L[A]$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'committed'

b. $@status_{L1}[A](A) \leftarrow$ 'committed'

3. abort A $(A \in act - \{U\})$

PRECONDITIONS:

a. $A \in @active_L[A]$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'aborted'

b. $@status_{L1}[A](A) \leftarrow$ 'aborted'

4. perform A.u $(A \in accesses(x), u \in values(x))$

PRECONDITIONS:

a. $A \in @active_L[x]$

b. $A \in prop\text{-}desc(V(x).holder)$

c. $u = V(x).value$

d. $anc(A) \cap @aborted_L[x] = \emptyset$

TRANSITIONS:

a. $status_{T1}(A) \leftarrow$ 'committed'

b. $@status_{L1}[x](A) \leftarrow$ 'committed'

c. $label_{T1}(A) \leftarrow u$

d. $data_{T1} \leftarrow data_T \cup \{(B,A): B \in datasteps_T(x)\} \cup \{(A,A)\}$

e. $V1(x,parent(A)) \leftarrow update(A)(u)$

5. <u>@create[β,α]</u> <u>A,d</u> ($A \in \text{act} - \{U\}$, $\beta,\alpha \in \text{loc}$, $d \subseteq \text{act}$)

PRECONDITIONS:

a. $A \in \text{@active}_L[\beta]$

b. $d = \text{@aborted}_L[\beta]$

TRANSITIONS:

a. $\text{@vertices}_{L1}[\alpha] \leftarrow \text{@vertices}_L[\alpha] \cup \{A\}$

b. $A \notin \text{@vertices}_L[\alpha] \Rightarrow \text{@status}_{L1}[\alpha](A) \leftarrow$ 'active'

c. $\text{@aborted}_{L1}[\alpha] \leftarrow \text{@aborted}_L[\alpha] \cup d$

d. $\alpha \in \text{obj},\ D \in d,\ B \in \text{desc}(D) \Rightarrow$
$V1(\alpha,B) \leftarrow \perp$

6. <u>@commit[β,α]</u> <u>A,d</u> ($A \in \text{act} - \{U\}$, $\beta,\alpha \in \text{loc}$, $d \subseteq \text{act}$)

PRECONDITIONS:

a. $A \in \text{@committed}_L[\beta]$

b. $d = \text{@aborted}_L[\beta]$

TRANSITIONS:

a. $\text{@vertices}_{L1}[\alpha] \leftarrow \text{@vertices}_L[\alpha] \cup \{A\}$

b. $\text{@status}_{L1}[\alpha](A) \leftarrow$ 'committed'

c. $\alpha \in \text{obj},\ V(\alpha,A) \neq \perp \Rightarrow$
$V1(\alpha,A) \leftarrow \perp$
$V1(\alpha,\text{parent}(A)) \leftarrow V(\alpha,A)$

d. $\text{@aborted}_{L1}[\alpha] \leftarrow \text{@aborted}_L[\alpha] \cup d$

e. $\alpha \in \text{obj},\ D \in d,\ B \in \text{desc}(D) \Rightarrow$
$V1(\alpha,B) \leftarrow \perp$

7. <u>@abort[β,α]</u> <u>A</u> ($A \in \text{act} - \{U\}$, $\beta,\alpha \in \text{loc}$)

PRECONDITIONS:

a. $A \in @\mathrm{aborted}_L[\beta]$

TRANSITIONS:

a. $@\mathrm{vertices}_{L1}[\alpha] \leftarrow @\mathrm{vertices}_L[\alpha] \cup \{A\}$

b. $@\mathrm{status}_{L1}[\alpha](A) \leftarrow$ 'aborted'

c. $\alpha \in \mathrm{obj},\ B \in \mathrm{desc}(A) \Rightarrow$
$V1(\alpha,B) \leftarrow \perp$

The preconditions and transitions for all local events are identical in L5 and L4 (except that the non-local orphan detection precondition, P4.4e, is eliminated at Level 5). Communications events are fundamentally different at Level 5, since orphan detection information is explicitly passed between locations with create and commit messages.

The orphan information that we include with create and commit messages is quite simple: a sending location must piggyback all the aborts it knows about onto these messages. These messages now correspond closely to the create and commit messages of the simplified orphan detection algorithm that we presented in Chapter 1. The "known aborts set" in these messages models the "DONE" list in the messages of this algorithm. While we show below that this information is *sufficient* (because there is a possibilities map from L5 to L4), other choices are possible. As a simple example, we conjecture that it is only necessary to send a *covering subset* of the known aborts in create and commit messages, because such a subset captures the same information about potential orphans. We have not attempted to take such optimizations into account, and we have focused on simplicity of description for our model. In general, at every level of our algebra hierarchy we make additional choices about the details of our model, and we further restrict the possible implementations which fit this model.

In our Level 5 model we do not piggyback the known aborts set onto *abort* messages. We can explain the difference between abort messages and create or commit messages by recalling (from Chapter 3) that in our idealized transaction system, aborts transfer no information (other than the fact that the abort occurred). Because the receiver of an abort message does not "learn anything" about the execution history, the sender need not tell the receiver about all potential orphans. Internal consistency is achieved by *coupling* the flow of normal information with the flow of orphan information. (In this case "orphan information" is just the set of known aborts at the sender.) Of course, it would not *hurt* to piggyback known aborts onto abort messages, and this additional information might allow some orphans to be detected sooner.

9.2 Specification of Mapping h_{54}

We define a (single-state) mapping from L5 to L4, h_{54}: L5 → L4. (We abbreviate "h_{54}" as "h" in this chapter.)

State Mapping

h: $\Sigma_5 \to \Sigma_4$ is the identity mapping: $h(\langle T,L,V\rangle) = \langle T,L,V\rangle \ \ \forall \langle T,L,V\rangle \in \Sigma_5$. Thus h fixes $\langle T,L,V\rangle$.

Event Mapping

h: $\mathcal{E}_5 \to \mathcal{E}_4^*$ is defined as follows. Let *ord4* be an arbitrary total order on $\mathcal{E}_4$, and let aborts-in(d) = {@abort[α] D: D $\in$ d}.

$$\begin{array}{lll}
\text{h: create A} & \to & \text{create A} \\
\text{commit A} & \to & \text{commit A} \\
\text{abort A} & \to & \text{abort A} \\
\text{perform A,u} & \to & \text{perform A,u}
\end{array}$$

$$\begin{array}{lll}
\text{@create}[\beta,\alpha]\text{ A,d} & \to & \text{@create}[\alpha]\text{ A} \bullet \langle\langle \text{aborts-in(d); } \textit{ord4}\rangle\rangle \\
\text{@commit}[\beta,\alpha]\text{ A,d} & \to & \text{@commit}[\alpha]\text{ A} \bullet \langle\langle \text{aborts-in(d); } \textit{ord4}\rangle\rangle \\
\text{@abort}[\beta,\alpha]\text{ A} & \to & \text{@abort}[\alpha]\text{ A}
\end{array}$$

Note that we map communications events @create and @commit into a sequence of events at Level 4. This sequence first creates or commits the primary action in the message, and then effectively aborts all actions in the aborts list, d. We will show that the order in which these abort events occur is unimportant; thus we let *ord4* be an arbitrary order.

9.3 Level 5 Invariants

Before stating the Level 5 invariants, we state two preliminary lemmas which will be used below:

Lemma 9.3.1: Let $\langle T,L,V\rangle \in \mathfrak{R}_5 \cap PRE_5(e)$, and $\langle T',L',V'\rangle = \langle T,L,V\rangle e$. Suppose that $\langle T,L,V\rangle$ satisfies I3, and $(\langle T,L,V\rangle,\langle T',L',V'\rangle)$ satisfies Ja and J3. If $ABORTS_T(A) \leq$ $@aborted_L[\alpha]$, and $A \in @committed_L[\alpha]$, then

$ABORTS_{T'}(A) \leq @aborted_{L'}[\alpha]$.

Proof: If $A \in @committed_L[\alpha]$, then by Lemma 7.3.2f, $A \in committed_T$,
$\Rightarrow ABORTS_{T'}(A) \leq ABORTS_T(A)$, by Lemma 6.3.3.3.

But $@aborted_L[\alpha] \subseteq @aborted_{L'}[\alpha]$, by Lemma 7.3.1a $\Rightarrow$ $@aborted_L[\alpha] \leq @aborted_{L'}[\alpha]$, by Lemma 2.2.1.1a.

And $ABORTS_T(A) \leq @aborted_L[\alpha]$, by hypothesis.

Thus $ABORTS_{T'}(A) \leq @aborted_{L'}[\alpha]$, by transitivity of $\leq$. ▮

Lemma 9.3.2: Let $\langle T,L,V\rangle \in \mathfrak{R}_5 \cap PRE_5(e)$, and $\langle T',L',V'\rangle = \langle T,L,V\rangle e$. Suppose that $\langle T,L,V\rangle$ satisfies I3, and $(\langle T,L,V\rangle,\langle T',L',V'\rangle)$ satisfies Ja and J3. If $SEQ\text{-}ABORTS_T(A) \leq$ $@aborted_L[\alpha]$, and $A \in @active_L[\alpha]$, then

$SEQ\text{-}ABORTS_{T'}(A) \leq @aborted_{L'}[\alpha]$.

Proof: If $A \in @active_L[\alpha]$, then by Lemma 7.3.2f, $A \in vertices_T$,
$\Rightarrow SEQ\text{-}ABORTS_{T'}(A) \leq SEQ\text{-}ABORTS_T(A)$, by Lemma 6.3.3.4.

But $@aborted_L[\alpha] \subseteq @aborted_{L'}[\alpha]$, by Lemma 7.3.1a $\Rightarrow$ $@aborted_L[\alpha] \leq @aborted_{L'}[\alpha]$, by Lemma 2.2.1.1a.

And $SEQ\text{-}ABORTS_T(A) \leq @aborted_L[\alpha]$, by hypothesis.

Thus $SEQ\text{-}ABORTS_{T'}(A) \leq @aborted_{L'}[\alpha]$, by transitivity of $\leq$. ▮

The following invariants are our key result for Level 5. They express the fact that the local states have the proper abort information at all times. We show these invariants relative to mapping h: Since h fixes $\langle T,L,V \rangle$, all invariants and pair-invariants in L4 can be applied to the proofs (by Lemma 4.2.4.3.6). (Recall that we have shown in Lemma 8.4.6 that I4, I3, and Ia are invariant in L4, and J3 and Ja are pair-invariant in L4.)

Lemma 9.3.3: Let $\langle T,L,V \rangle \in \mathcal{R}_5$. Then the following are invariant relative to h:

$(\forall \alpha \in \text{loc})$

a. $A \in @\text{committed}_L[\alpha] \Rightarrow \text{ABORTS}_T(A) \leq @\text{aborted}_L[\alpha]$

b. $A \in @\text{active}_L[\alpha] \Rightarrow \text{SEQ-ABORTS}_T(A) \leq @\text{aborted}_L[\alpha]$

Proof: It is trivial to show that (a),(b) are 0-invariant (i.e. that they hold for σ_5): (a) holds vacuously, and for (b) only $U \in @\text{vertices}_{L_0}[\alpha]$, but $\text{SEQ-ABORTS}_{T_0} = \emptyset$.

For the induction step, let $\langle T,L,V \rangle \in \mathcal{R}_5 \cap \text{PRE}_5(e)$, and assume that (a),(b) hold for $\langle T,L,V \rangle$. Let $\langle T',L',V' \rangle = \langle T,L,V \rangle e$. We must show that (a),(b) hold for $\langle T',L',V' \rangle$. By Lemma 4.2.4.3.6, we can assume that $\langle T,L,V \rangle$ and $\langle T',L',V' \rangle$ satisfy any invariants from I4, I3 or Ia, and we can assume that $(\langle T,L,V \rangle,\langle T',L',V' \rangle)$ satisfy any pair-invariants from J3 or Ja.

Using the Induction Hypothesis, and invariants I3, J3, Ja, we can apply Lemmas 9.3.1 and 9.3.2 to conclude that

$A \in @\text{committed}_L[\alpha] \Rightarrow \text{ABORTS}_{T'}(A) \leq @\text{aborted}_{L'}[\alpha]$.

$A \in @\text{active}_L[\alpha] \Rightarrow \text{SEQ-ABORTS}_{T'}(A) \leq @\text{aborted}_{L'}[\alpha]$.

Thus we need only show that (a) holds (respectively, (b) holds) for $\langle T',L',V' \rangle$ where $A \in @\text{committed}_{L'}[\alpha] - @\text{committed}_L[\alpha]$ (respectively, $A \in @\text{active}_{L'}[\alpha] - @\text{active}_L[\alpha]$). We consider all possible events, e, for these remaining cases:

1. create A (note that $A \neq U$)

$@\text{committed}_{L'}[\alpha] = @\text{committed}_L[\alpha]$.
$@\text{active}_{L'}[\alpha] - @\text{active}_L[\alpha] = \{A\}$, for $\alpha = \text{creator}(A)$.

$@\text{active}_{L'}[\alpha] = @\text{active}_{L}[\alpha]$, for $\alpha \neq \text{creator}(A)$.

a. Holds vacuously.

b. (We need only consider $\alpha = \text{creator}(A)$.)
By P5.1b, parent(A) $\in @\text{active}_{L}[\alpha]$,
$\Rightarrow$ SEQ-ABORTS$_{T'}$(parent(A)) $\leq @\text{aborted}_{L'}[\alpha]$.

By P5.1c, $(B,A) \in \text{seq}$, $B \neq A \Rightarrow B \in @\text{done}_{L}[\alpha]$. Thus $B \in$ v-seq$_{T'}$(A) $\Rightarrow B \in @\text{committed}_{L}[\alpha]$,
$\Rightarrow$ ABORTS$_{T'}$(B) $\leq @\text{aborted}_{L'}[\alpha]$.

$B \in$ i-seq$_{T'}$(A) $\Rightarrow B \in @\text{aborted}_{L}[\alpha] \Rightarrow B \in @\text{aborted}_{L'}[\alpha]$,
$\Rightarrow$ i-seq$_{T'}$(A) $\leq @\text{aborted}_{L'}[\alpha]$.

Thus SEQ-ABORTS$_{T'}$(parent(A)) $\cup \bigcup_{B \in \text{v-seq}_{T'}(A)} \text{ABORTS}_{T'}(B)$
$\cup$ i-seq$_{T'}$(A) $\leq @\text{aborted}_{L'}[\alpha]$, by Lemma 2.2.1.1c.

Thus SEQ-ABORTS$_{T'}$(A) $\leq @\text{aborted}_{L'}[\alpha]$, by Lemma 6.2.1.4.

2. commit A (note A $\notin$ accesses)

$@\text{committed}_{L'}[\alpha] - @\text{committed}_{L}[\alpha] = \{A\}$, for $\alpha = A$.
$@\text{committed}_{L'}[\alpha] = @\text{committed}_{L}[\alpha]$, for $\alpha \neq A$.
$@\text{active}_{L'}[\alpha] \subseteq @\text{active}_{L}[\alpha]$.

a. (We need only consider $\alpha = A$.)
ABORTS$_{T'}$(A) = i-precedes$_{T'}$(A) $\cup \bigcup_{B \in \text{v-precedes}_{T'}(A)} \text{ABORTS}_{T'}(B)$.
Since A $\notin$ accesses, v-data-anc$_{T'}$(A) = i-data-anc$_{T'}$(A) = $\emptyset$; thus v-precedes$_{T'}$(A) = v-anc-seq$_{T'}$(A) $\cup$ v-child$_{T'}$(A), and i-precedes$_{T'}$(A) = i-anc-seq$_{T'}$(A) $\cup$ i-child$_{T'}$(A).

Thus ABORTS$_{T'}$(A) = i-anc-seq$_{T'}$(A) $\cup \bigcup_{B \in \text{v-anc-seq}_{T'}(A)} \text{ABORTS}_{T'}(B)$ $\cup$
i-child$_{T'}$(A) $\cup \bigcup_{B \in \text{v-child}_{T'}(A)} \text{ABORTS}_{T'}(B)$.
= SEQ-ABORTS$_{T'}$(A) $\cup$ i-child$_{T'}$(A) $\cup \bigcup_{B \in \text{v-child}_{T'}(A)} \text{ABORTS}_{T'}(B)$.
But $A \in @\text{active}_{L}[A]$ by P5.2a, $\Rightarrow$ SEQ-ABORTS$_{T'}$(A) $\leq @\text{aborted}_{L'}[A]$, by Lemma 9.3.2.

If $B \in$ children$_{T'}$(A), then $B \in$ children$_{T}$(A). But by Lemma 7.3.2a, $B \in$ children$_{T}$(A) $\Rightarrow B \in @\text{vertices}_{L}[A]$. By P5.2b, $@\text{children}_{L}[A](A) \subseteq @\text{done}_{L}[A]$. From Lemma 7.3.2f, it follows that v-child$_{T'}$(A) $\subseteq @\text{committed}_{L}[A]$, and i-child$_{T'}$(A) $\subseteq$

$@\text{aborted}_L[A]$.

Thus $B \in \text{v-child}_{T'}(A) \Rightarrow \text{ABORTS}_{T'}(B) \leq @\text{aborted}_{L'}[A]$, by Lemma 9.3.1.

$B \in \text{i-child}_{T'}(A) \Rightarrow B \in @\text{aborted}_L[A] \Rightarrow B \in @\text{aborted}_{L'}[A]$, by Lemma 7.3.1a.

Thus we have shown
$\text{SEQ-ABORTS}_{T'}(A) \leq @\text{aborted}_{L'}[A]$,
$\text{i-child}_{T'}(A) \leq @\text{aborted}_{L'}[A]$, and
$\bigcup_{B \in \text{v-child}_{T'}(A)} \text{ABORTS}_{T'}(B) \leq @\text{aborted}_{L'}[A]$.
$\text{ABORTS}_{T'}(A) \leq @\text{aborted}_{L'}[A]$ follows directly from Lemma 2.2.1.1c.

b. Holds vacuously.

3. abort A

$@\text{committed}_{L'}[\alpha] = @\text{committed}_L[\alpha]$
$@\text{active}_{L'}[\alpha] \subseteq @\text{active}_L[\alpha]$

a. Holds vacuously.

b. Holds vacuously.

4. perform A,u

$@\text{committed}_{L'}[\alpha] - @\text{committed}_L[\alpha] = \{A\}$, for $\alpha = x$ $(x = \text{object}(A))$
$@\text{committed}_{L'}[\alpha] = @\text{committed}_L[\alpha]$, for $\alpha \neq x$
$@\text{active}_{L'}[\alpha] \subseteq @\text{active}_L[\alpha]$

a. (We need only consider $\alpha = x$)
$\text{ABORTS}_{T'}(A) = \text{i-precedes}_{T'}(A) \cup \bigcup_{B \in \text{v-precedes}_{T'}(A)} \text{ABORTS}_{T'}(B)$.
Since $A \in$ accesses, $\text{v-child}_{T'}(A) = \text{i-child}_{T'}(A) = \emptyset$; thus $\text{v-precedes}_{T'}(A) = \text{v-anc-seq}_{T'}(A) \cup \text{v-data-anc}_{T'}(A)$, and $\text{i-precedes}_{T'}(A) = \text{i-anc-seq}_{T'}(A) \cup \text{i-data-anc}_{T'}(A)$.

Thus $\text{ABORTS}_{T'}(A) = \text{i-anc-seq}_{T'}(A) \cup \bigcup_{B \in \text{v-anc-seq}_{T'}(A)} \text{ABORTS}_{T'}(B) \cup$
$\text{i-data-anc}_{T'}(A) \cup \bigcup_{B \in \text{v-data-anc}_{T'}(A)} \text{ABORTS}_{T'}(B)$.
$= \text{SEQ-ABORTS}_{T'}(A) \cup \text{i-data-anc}_{T'}(A) \cup \bigcup_{B \in \text{v-data-anc}_{T'}(A)} \text{ABORTS}_{T'}(B)$.

But A $\in$ @active$_L$[x] by P5.4a, $\Rightarrow$ SEQ-ABORTS$_{T'}$(A) $\leq$ @aborted$_{L'}$[x], by Lemma 9.3.2.

If (B,A) $\in$ v-data$_{T'}$, then B $\in$ @visible$_{L'}$[x](A), by Lemma 7.3.2j, $\Rightarrow$ B $\in$ @visible$_L$[x](A).
Thus A$\downarrow$B $\in$ @committed$_L$[x], $\Rightarrow$ ABORTS$_{T'}$(A$\downarrow$B) $\leq$ @aborted$_{L'}$[x], by Lemma 9.3.1.

If (B,A) $\in$ i-data$_{T'}$, then B $\in$ @dead$_{L'}$[x], by Lemma 7.3.2k. But B $\in$ @dead$_{L'}$[x] $\Rightarrow$ {crucial$_{T'}$(B)} $\leq$ @aborted$_{L'}$[x], by Lemma 7.3.2l.

Thus we have shown
SEQ-ABORTS$_{T'}$(A) $\leq$ @aborted$_{L'}$[x],
i-data-anc$_{T'}$(A) $\leq$ @aborted$_{L'}$[x], and
$\bigcup_{B \in \text{v-data-anc}_{T'}(A)} \text{ABORTS}_{T'}(B) \leq$ @aborted$_{L'}$[x].
ABORTS$_{T'}$(A) $\leq$ @aborted$_{L'}$[x] follows directly from Lemma 2.2.1.1c.

b. Holds vacuously.

5. @create[β,α] A,d

@committed$_{L'}$[γ] = @committed$_L$[γ].
@vertices$_{L'}$[γ] = @vertices$_L$[γ] $\cup$ {A}, for $\gamma = \alpha$ (unchanged for all other locations).
@active$_{L'}$[α] - @active$_L$[α] $\subseteq$ {A} (might be $\emptyset$).
@active$_{L'}$[γ] = @active$_L$[γ], for $\gamma \neq \alpha$.

a. Holds vacuously.

b. (We need only consider $\gamma = \alpha$.)
By P5.5a, A $\in$ @active$_L$[β]; thus SEQ-ABORTS$_T$(A) $\leq$ @aborted$_L$[β], by Lemma 9.3.1.

But d = @aborted$_L$[β] by P5.5b, and d $\subseteq$ @aborted$_{L'}$[α] (by T5.5c). Thus SEQ-ABORTS$_T$(A) $\leq$ @aborted$_{L'}$[α].

But A $\in$ @active$_L$[β] $\Rightarrow$ A $\in$ vertices$_T$ by Lemma 7.3.2f, $\Rightarrow$ SEQ-ABORTS$_{T'}$(A) $\leq$ SEQ-ABORTS$_T$(A), by Lemma 6.3.3.4.

Thus SEQ-ABORTS$_{T'}$(A) $\leq$ @aborted$_{L'}$[α], by transitivity of $\leq$.

6. @commit[β,α] A,d

   @committed$_{L'}$[α] - @committed$_L$[α] $\subseteq$ {A}.
   @committed$_{L'}$[γ] = @committed$_L$[γ], for $\gamma \neq \alpha$.
   @active$_{L'}$[γ] $\subseteq$ @active$_L$[γ].

   a. (We need only consider $\gamma = \alpha$.)

      By P5.6a, A $\in$ @committed$_L$[β]; thus ABORTS$_T$(A) $\leq$ @aborted$_L$[β], by Lemma 9.3.1.

      But d = @aborted$_L$[β] by P5.6b, and d $\subseteq$ @aborted$_{L'}$[α] (by T5.5d). Thus ABORTS$_T$(A) $\leq$ @aborted$_{L'}$[α].

      But A $\in$ @committed$_L$[β] $\Rightarrow$ A $\in$ committed$_T$ by Lemma 7.3.2f, $\Rightarrow$ ABORTS$_{T'}$(A) $\leq$ ABORTS$_T$(A), by Lemma 6.3.3.3.

      Thus ABORTS$_{T'}$(A) $\leq$ @aborted$_{L'}$[α], by transitivity of $\leq$.

   b. Holds vacuously.

7. @abort[β,α] A

   @committed$_{L'}$[γ] = @committed$_L$[γ].
   @active$_{L'}$[γ] $\subseteq$ @active$_L$[γ].

   a. Holds vacuously.

   b. Holds vacuously. ▮

9.4 Proof of Possibilities Map for h_{54}

We now show that h is a possibilities map. Let I5 be the conjunction of all properties in Lemma 9.3.3. We will show that h is a possibilities map relative to I5.

Lemma 9.4.1: h preserves initial states.

Proof: Immediate since h(<T$_0$,L$_0$,V$_0$>) = <T$_0$,L$_0$,V$_0$>. ▮

Lemma 9.4.2: h preserves transitions relative to I5.

Proof: We must show that if $\langle T,L,V\rangle \in \mathrm{PRE}_5(e) \cap \mathfrak{R}_5 \cap \mathrm{I5}$, and $h(\langle T,L,V\rangle) \in \mathrm{PRE}_4(h(e)) \cap \mathfrak{R}_4$, then $h(\langle T,L,V\rangle e) = h(\langle T,L,V\rangle)h(e)$.

But $h(\langle T,L,V\rangle) = \langle T,L,V\rangle$, so we must show the following:

Let $\langle T,L,V\rangle \in \mathrm{PRE}_5(e) \cap \mathfrak{R}_5 \cap \mathrm{PRE}_4(h(e)) \cap \mathfrak{R}_4$.
Let $\langle T,L,V\rangle e = \langle T1,L1,V1\rangle$ (in L5), $\langle T2,L2,V2\rangle = \langle T,L,V\rangle h(e)$ (in L4).
Then $\langle T1,L1,V1\rangle = \langle T2,L2,V2\rangle$

For the local steps (create, commit, abort, perform), h(e) = e, and it is easily verified by inspection that the effects of these events on T, L, and V are identical in L5 and L4. It is also easily verified by inspection that the effect of @abort[β,α] A is identical to the effect of @abort[α] A.

For communications events @create and @commit, transition steps T5.5a,b, and T5.6a,b,c are identical to transition effects T4.5a,b, and T4,5a,b,c, respectively. Transition steps T5.5c,d, and T5.6d,e, respectively, accomplish the same effect as the sequence of aborts <<aborts-in(d); *ord4*>>: Adding all aborts in d to @aborted$_L$[α] (Level 5) has the same effect as adding them individually (Level 4). To see that updating of value maps is also preserved, note that an abort at an object removes all descendants of the aborted action from the value map at that object. But individually removing descendants of each action (Level 4) has the same effect as removing all descendants from actions in d at once (Level 5). Note that this removal of descendants is clearly commutative, and thus the order of abort steps in aborts-in(d) makes no difference. ∎

Lemma 9.4.3: h preserves preconditions relative to I5.

Proof: We must show that if $\langle T,L,V\rangle \in \mathrm{PRE}_5(e) \cap \mathfrak{R}_5 \cap \mathrm{I5}$, and $h(\langle T,L,V\rangle) \in \mathfrak{R}_4$, then $h(\langle T,L,V\rangle) \in \mathrm{PRE}_4(h(e))$.

Since $h(\langle T,L,V\rangle) = \langle T,L,V\rangle$, we show
$\langle T,L,V\rangle \in \mathrm{PRE}_5(e) \cap \mathfrak{R}_5 \cap \mathrm{I5} \cap \mathfrak{R}_4 \Rightarrow \langle T,L,V\rangle \in \mathrm{PRE}_4(h(e))$.

Preservation of preconditions is easily verified by inspection for all *local* steps other than perform, since preconditions are identical in L5 and L4. We prove preservation of preconditions for event e = perform A,u, and for the communications steps:

1. perform A,u

   a. P5.4a $\Leftrightarrow$ P4.4a.

   b. P5.4b $\Leftrightarrow$ P4.4b.

   c. P5.4c $\Leftrightarrow$ P4.4c.

   d. P5.4d $\Leftrightarrow$ P4.4d.

   e. $B \in @\text{visible}_L[x](A,x) \Rightarrow A{\downarrow}B \in @\text{committed}_L[x]$
   $\Rightarrow \text{ABORTS}_T(A{\downarrow}B) \leq @\text{aborted}_L[x]$ by Lemma 9.3.3a.

   But by P5.4d, $\text{anc}(A) \cap @\text{aborted}_L[x] = \emptyset$. Thus $\text{anc}(A) \cap \text{ABORTS}_T(A{\downarrow}B) = \emptyset$, by Lemma 2.2.1.1d.

2. $e = @\text{create}[\beta,\alpha]$ A,d
   $h(e) = @\text{create}[\alpha]\ A \bullet \langle\langle \text{aborts-in}(d);\ \mathit{ord4} \rangle\rangle$

   First we show that $\langle T,L,V \rangle \in \text{PRE}_4(@\text{create}[\alpha]\ A)$:

   a. P5.5a $\Rightarrow$ $A \in @\text{active}_L[\beta]$, $\Rightarrow$ $A \in @\text{vertices}_L[\beta]$, which automatically satisfies P4.5a. (P4.5a requires that there be *some* β for which $A \in @\text{vertices}_L[\beta]$.)

   Now let e' be the prefix of h(e) preceding $@\text{abort}[\alpha]\ D$ (where $D \in d$), and let $\langle T1,L1,V1 \rangle = \langle T,L,V \rangle e'$ (in L4). We show that $\langle T1,L1,V1 \rangle \in \text{PRE}_4(@\text{abort}[\alpha]\ D)$:

   a. $D \in d \Rightarrow D \in @\text{aborted}_L[\beta] \Rightarrow D \in \text{aborted}_T$ by Lemma 7.3.2f,
   $\Rightarrow D \in @\text{aborted}_L[D]$ by Lemma 7.3.2c,
   $\Rightarrow D \in @\text{aborted}_{L1}[D]$ by Lemma 7.3.1a.

3. $@\text{commit}[\beta,\alpha]$ A,d $e = @\text{commit}[\beta,\alpha]$ A,d
   $h(e) = @\text{commit}[\alpha]\ A \bullet \langle\langle \text{aborts-in}(d);\ \mathit{ord4} \rangle\rangle$

   First we show that $\langle T,L,V \rangle \in \text{PRE}_4(@\text{commit}[\alpha]\ A)$:

   a. P5.6a $\Rightarrow$ $A \in @\text{committed}_L[\beta]$, which satisfies P4.6a.

   Now let e' be the prefix of h(e) preceding $@\text{abort}[\alpha]\ D$ (where $D \in d$), and let $\langle T1,L1,V1 \rangle = \langle T,L,V \rangle e'$ (in L4). We show that $\langle T1,L1,V1 \rangle \in \text{PRE}_4(@\text{abort}[\alpha]$

D):

a. $D \in d \Rightarrow D \in @aborted_L[\beta] \Rightarrow D \in aborted_T$ by Lemma 7.3.2f,
$\Rightarrow D \in @aborted_L[D]$ by Lemma 7.3.2c,
$\Rightarrow D \in @aborted_{L1}[D]$ by Lemma 7.3.1a.

4. @abort[β,α] A

a. P5.7a $\Rightarrow A \in @aborted_L[\beta]$, which satisfies P4.7a. ▮

Lemma 9.4.4: h is a possibilities map relative to I5.

Proof: Follows immediately from Lemmas 9.4.1, 9.4.2, 9.4.3, and from Lemma 4.2.4.2.4. ▮

Theorem 9.4.5: h is a possibilities map, and I5 is invariant in L5.

Proof: By Lemma 9.3.3, I5 is invariant relative to h. By Lemma 9.4.4, h is a possibilities map relative to I5. We apply Lemma 4.2.4.2.6 to conclude that h is a possibilities map, and I5 is an invariant. ▮

Since h_{54} is a possibilities map which fixes <T,L,V>, all invariants and pair-invariants from L4 carry down to L5. We summarize the invariants for L5 as follows:

Lemma 9.4.6: Ia, I3, I4, and I5 are invariant in L4, and Ja, J3 are pair-invariant in L4.

Proof: Invariance of I5 is shown in Theorem 9.4.5. Since h_{54} is a possibilities map which fixes <T,L,V>, and Ia, I3, and I4 are invariant in L3, Ia, I3, and I4 are invariant in L4, by Lemma 4.2.4.3.5. Similarly since Ja, J3 are pair-invariant in L4, Ja and J3 are pair-invariant in L5, by Lemma 4.2.4.3.5. ▮

9.5 Level 6 Algebra and Mapping h_{65}

At Level 6 we remove the global action tree, T. Since we have localized all preconditions in Level 5, the global tree can now be properly regarded as a "virtual" component of the state.

$L6 = (\mathcal{E}_6, \Sigma_6, \sigma_6, \tau_6)$

$\Sigma_6 = \{\langle L,V \rangle\}$, where the components are:

L - local UAS's (as in L3)
V - value maps (as in L4)

$\sigma_6 = \langle L_0, V_0 \rangle$

L_0, V_0 - as in L4

$\mathcal{E}_6 = \mathcal{E}_5$

τ_6 is identical to τ_5, except that all transitions involving T are discarded (T5.1a,b, T5.2a, T5.3a, T5.4a,c,d).

Let h_{65}: L6 → L5 be the augmentation map from Level 6 to Level 5 (Definition 4.2.5.1). Thus h_{65} is the identity map on events, and the state mapping maps $\langle L,V \rangle$ to all possible states $\langle T,L,V \rangle$ in Σ_5.

Theorem 9.5.1: h_{65} is a possibilities map.

Proof: Follows immediately from Lemma 4.2.5.3. ∎

By Lemma 4.2.5.2, h_{65} fixes $\langle L,V \rangle$, so all invariants and pair-invariants for L and V from L5 carry down to L6. Most of these properties involve T, but all invariants from I4 except for 8.3.1e do not involve T, nor do the pair-invariants J3. We summarize these invariants and pair-invariants for L6 in the following Lemma. Let I4' denote I4 with 8.3.1e removed. (Thus I4' is just all invariants from I4 which apply to the local state $\langle L,V \rangle$.)

Lemma 9.5.2: I4' is invariant for L6, and J3 is pair-invariant in L6.

Proof: Since h_{65} is a possibilities map which fixes $\langle L,V \rangle$, and I4' is invariant for $\langle L,V \rangle$ in L5, I4' is invariant in L6, by Lemma 4.2.4.3.5. Similarly since J3 is pair-invariant for $\langle L,V \rangle$ in L5, J3 is pair-invariant in L6, by Lemma 4.2.4.3.5. ▮

10. Distributed System Model

Level 7 is our lowest-level model of the transaction system. At this level we partition the system state among *nodes*, and we use a communications model which takes into account arbitrary delays in message delivery. This model is a message-based distributed event-state algebra as described in Chapter 4. Nodes communicate by sending and receiving *messages* via a *message buffer.*

We require that each object and each action reside at a particular node (its "home node"). A node's state consists of a UAS and a value map for each object which resides there. We can thus view nodes as a grouping structure for the "tree locations" from Level 6. The mapping from node states (Level 7) to local states at tree locations (Level 6) is a straightforward "explosion" of the node states. Similarly the Level 6 value map can be constructed from the value maps at each node.

The only complexity in mapping from Level 7 to Level 6 is in modeling the communications delays at Level 7, since the communications events at Level 6 are "instantaneous." We resolve this discrepancy by treating messages themselves as locations. We regard a message as an initially empty "slot" for information; once this message is sent, the slot is filled. Since messages are never removed from the message buffer in our Level 7 model, it is natural to regard this message slot as a "location" at Level 6. The communications delay at Level 7 is explained at Level 6 by imagining that all messages are instantaneous, but that they are sent indirectly via another location (the message slot).

10.1 Level 7 Algebra

$L7 = (\mathcal{E}_7, \Sigma_7, \sigma_7, \tau_7)$

The Level 7 Algebra is a message-based algebra as defined in Chapter 4 (Definition 4.3.2.1). Let Nodes = {1,2,...,n} name the nodes in the system, and let "buf" name the message buffer. We will use $\Gamma = \Sigma_7$ in this chapter, so that we can subscript the state space without confusing these subspaces with the state spaces of higher levels in our algebra hierarchy.

We assume that each object in the system resides at a particular node, and each action runs at a single node. We call this node the home node of the action or object. Formally,

home: tloc → Nodes. (If $A \in$ accesses, then we will use home(A) synonymously with home(object(A)).

Let $obj(i) = \{x \in obj: home(x) = i\}$,

$act(i) = \{A \in act - \{U\} - accesses: home(A) = i\}$,

$tloc(i) = obj(i) \cup act(i)$.

State Space:

The local state at a node consists of a UAS for the node together with a "local" value map for each object whose home is at that node:

$\Gamma_i = \{\langle l,v\rangle: l \in UAS$, and $v: obj(i) \times act \rightarrow values(obj) \cup \{\perp\}\}$, where $i \in Nodes$.

If $D \in \Gamma$ and $i \in Nodes$, then we denote the UAS and value map components of D.i by D.i.l and D.i.v, respectively. We extend the definitions of V(x), V(x).holder, V(x).value, etc., from value maps to "local value maps" in the obvious way.

The set of messages is defined as follows:

$$\begin{aligned} Msgs = &\ \{\#create(i,j)\ A,d: i,j \in Nodes, A \in act - \{U\}, d \subseteq act\} \\ \cup &\ \{\#commit(i,j)\ A,d: i,j \in Nodes, A \in act - \{U\} - accesses, d \subseteq act\} \\ \cup &\ \{\#abort(i,j)\ A: i,j \in Nodes, A \in act - \{U\}\} \end{aligned}$$

The message buffer space is $\Gamma_{buf} = \mathcal{P}(Msgs)$.

If $D \in \Gamma$, and $i \in Nodes$, then we abbreviate any function $prop_{D.i.l}$ by $\#prop_D[i]$. (This notation is similar to the notation introduced for locations, but note that i is now a *node* rather than a location.)

Initial State:

$\sigma_7 = D_0$, where D_0 is defined by

$D_0.buf = \emptyset$,

$\forall i \in Nodes$, $D_0.i.l = T_\mu$, the trivial UAS, and

$D_0.i.v(x,U) = init(x)$, $\forall x \in obj(i)$,

$D_0.i.v(x,A) = \perp$, $\forall A \neq U$.

The UAS and value map components of $D_0.i$ correspond in a natural way to L_0 and V_0.

Events:

$\mathcal{E}_7$ consists of local events (create, commit, abort, perform), and communications events **send M**, **receive M**, for M $\in$ Msgs. Local events are similar to the corresponding local events at Level 6.

At Level 7 we include a qualifier "(d)" on create, commit, and perform events. For example, a create event takes the form: create A (d), where d $\subseteq$ act. The preconditions for the create, commit and perform events requires that "d" be the set of known aborts at the node where the event occurs. "d" does not enter into any transitions. We can thus regard "(d)" as *recording* the set of known aborts when the event occurs; including this qualifier does not change the semantics of the events. The qualifier "(d)" is useful when we construct a mapping from Level 7 to Level 6: "local" events at Level 7 will map into a local event at Level 6 plus a sequence of communications events at Level 6. (Conceptually in this mapping we regard the occurrence of an event at a *node* as an occurrence at a single location at that node, followed by a broadcast of the event (with Level 6 communications events) to all other locations at that node. Of course, at Level 7 no "real" communications events occur.) Because these Level 6 communications events require a "done" list, we extract it from the "(d)" in the Level 7 event.

(This device of qualifying events with a part of the state allows us to construct an *event-homomorphic* mapping between algebras. If the qualifier were not used, then the proper mapping from a lower-level event to the higher-level sequence of events would depend on the lower-level *state* as well as on the lower-level event, i.e. the event mapping would not be event-homomorphic.)

Transition Relation

Although Definition 4.3.1.1 describes the total transition relation of a message-based algebra in terms of *local* transition relations for each component, we will not describe local transition relations individually. Instead we present the total transition relation. It should be clear that preconditions and effects are properly localized (i.e. the local transition relations could be constructed easily from our total transition relation).

Let e $\in \mathcal{E}_7$, D $\in \Gamma$, De = D1.

1. **create A (d)** (A $\in$ act - {U}, home(creator(A)) = i, d $\subseteq$ act)

PRECONDITIONS:

a. $A \notin \#\mathrm{vertices}_D[i]$

b. $\mathrm{parent}(A) \in \#\mathrm{active}_D[i]$

c. $(B,A) \in \mathrm{seq},\ B \neq A \implies B \in \#\mathrm{done}_D[i]$

d. $d = \#\mathrm{aborted}_D[i]$

TRANSITIONS:

a. $\#\mathrm{vertices}_{D1}[i] \leftarrow \#\mathrm{vertices}_D[i] \cup \{A\}$

b. $\#\mathrm{status}_{D1}[i](A) \leftarrow$ 'active'

2. commit A (d) $(A \in \mathrm{act} - \{U\} - \mathrm{accesses},\ \mathrm{home}(A) = i,\ d \subseteq \mathrm{act})$

PRECONDITIONS:

a. $A \in \#\mathrm{active}_D[i]$

b. $\#\mathrm{children}_D[i](A) \subseteq \#\mathrm{done}_D[i]$

c. $d = \#\mathrm{aborted}_D[i]$

TRANSITIONS:

a. $\#\mathrm{status}_{D1}[i](A) \leftarrow$ 'committed'

b. $\forall x \in \mathrm{obj}(i),\ D.i.v(x,A) \neq \perp \implies$
$D1.i.v(x,A) \leftarrow \perp$
$D1.i.v(x,\mathrm{parent}(A)) \leftarrow D.i.v(x,A)$

3. abort A $(A \in \mathrm{act} - \{U\},\ \mathrm{home}(A) = i)$

PRECONDITIONS:

a. $A \in \#\mathrm{active}_D[i]$

TRANSITIONS:

a. $\#\mathrm{status}_{D1}[i](A) \leftarrow$ 'aborted'

b. $\forall x \in \mathrm{obj}(i),\ B \in \mathrm{desc}(A) \implies$
$D1.i.v(x,B) \leftarrow \perp$

4. perform A,u (d) $(A \in accesses(x), u \in values(x), home(x) = i, d \subseteq act)$

PRECONDITIONS:

a. $A \in \#active_D[i]$

b. $A \in prop\text{-}desc(D.i.v(x).holder)$

c. $u = D.i.v(x).value$

d. $anc(A) \cap \#aborted_D[i] = \emptyset$

e. $d = \#aborted_D[i]$

TRANSITIONS:

a. $\#status_{D1}[i](A) \leftarrow$ 'committed'

b. $D1.i.v(x,parent(A)) \leftarrow update(A)(u)$

5. send #create(i,j) A,d $(A \in act - \{U\}, i,j \in Nodes, d \subseteq act)$

PRECONDITIONS:

a. $A \in \#active_D[i]$

b. $d = \#aborted_D[i]$

TRANSITIONS:

a. $D1.buf \leftarrow D.buf \cup \{\#create(i,j)\ A,d\}$

6. receive #create(i,j) A,d $(A \in act - \{U\}, i,j \in Nodes, d \subseteq act)$

PRECONDITIONS:

a. $\#create(i,j)\ A,d \in D.buf$

TRANSITIONS:

a. $\#vertices_{D1}[j] \leftarrow \#vertices_D[j] \cup \{A\}$

b. $A \notin \#vertices_D[j] \Rightarrow \#status_{D1}[j](A) \leftarrow$ 'active'

c. $\#\text{aborted}_{D1}[j] \leftarrow \#\text{aborted}_{D}[j] \cup d$

d. $\forall x \in \text{obj}(j),\ C \in d,\ B \in \text{desc}(C) \Rightarrow$
$D1.j.v(x,B) \leftarrow \perp$

7. send #commit(i,j) A,d $(A \in \text{act} - \{U\},\ i,j \in \text{Nodes},\ d \subseteq \text{act})$

PRECONDITIONS:

a. $A \in \#\text{committed}_{D}[i]$

b. $d = \#\text{aborted}_{D}[i]$

TRANSITIONS:

a. $D1.buf \leftarrow D.buf \cup \#\text{commit}(i,j)\ A,d$

8. receive #commit(i,j) A,d $(A \in \text{act} - \{U\},\ i,j \in \text{Nodes},\ d \subseteq \text{act})$

PRECONDITIONS:

a. $\#\text{commit}(i,j)\ A,d \in D.buf$

TRANSITIONS:

a. $\#\text{vertices}_{D1}[j] \leftarrow \#\text{vertices}_{D}[j] \cup \{A\}$

b. $\#\text{status}_{D1}[j](A) \leftarrow$ 'committed'

c. $\forall x \in \text{obj}(j),\ D.j.v(x,A) \neq \perp \Rightarrow$
$D1.j.v(x,A) \leftarrow \perp$
$D1.j.v(x,\text{parent}(A)) \leftarrow D.j.v(x,A)$

d. $\#\text{aborted}_{D1}[j] \leftarrow \#\text{aborted}_{D}[j] \cup d$

e. $\forall x \in \text{obj}: \text{home}(x) = j,\ C \in d,\ B \in \text{desc}(C) \Rightarrow$
$D1.j.v(x,B) \leftarrow \perp$

9. send #abort(i,j) A $(A \in \text{act} - \{U\},\ i,j \in \text{Nodes})$

PRECONDITION:

a. $A \in \#\text{aborted}_{D}[i]$

TRANSITIONS:

a. $D1.buf \leftarrow D.buf \cup \ \#abort(i,j)$

10. <u>receive #abort(i,j) A</u> $(A \in act - \{U\},\ i,j \in Nodes)$

PRECONDITION:

a. $\#abort(i,j)\ A\ \in D.buf$

TRANSITIONS:

a. $\#vertices_{D1}[j] \leftarrow \#vertices_{D}[j] \cup \{A\}$

b. $\#status_{D1}[j](A) \leftarrow$ 'aborted'

c. $\forall x \in obj(j),\ B \in desc(A)\ \Rightarrow$
$D1.j.v(x,B) \leftarrow \bot$

10.2 Specification of Mapping h_{76}

We define a (single-state) mapping from L7 to L6, h_{76}: L7 → L6. (We abbreviate "h_{76}" as "h" in this chapter.)

At this point we instantiate the (previously unspecified) set of locations, loc; we define
$loc = tloc \cup Msgs$

We regard a message as a location because it is a container for information. The local information at this location is essentially the information contained in the message. As we explained above, we imagine that each message is a predefined "slot" for the particular combination of information that it represents. Originally this slot is empty; when the message is sent, the slot is filled.

<u>State Mapping</u>

h: $\Sigma_7 \rightarrow \Sigma_6$ is defined as follows. Let $D \in \Gamma$, $h(D) = \langle L,V \rangle$, then

$V = valuemap(D)$, where valuemap(D) is defined as $\{((o,a),u)\text{: } D.home(o).v(o,a) = u\}$.

Valuemap is defined exactly as we expect: the "total" valuemap for Level 6 is constructed by combining all local value maps. This mapping is so trivial that we can almost regard it as a simple change in notation.

L is defined by:

1. If $\alpha \in$ tloc, then $L(\alpha) = D.\mathrm{home}(\alpha).l$

2. If $\alpha \in$ Msgs, and $\alpha \notin$ D.buf, then $L(\alpha) = T_u$.

3. If $\alpha \in$ Msgs, and $\alpha \in$ D.buf, then

   a. If $\alpha = \#\mathrm{create}(i,j)\ A,d$, then $L(\alpha) = T$, where
   $\mathrm{vertices}_T = \{U,A\} \cup d$
   $\mathrm{committed}_T = \emptyset$
   $\mathrm{aborted}_T = d$

   b. If $\alpha = \#\mathrm{commit}(i,j)\ A,d$, then $L(\alpha) = T$, where
   $\mathrm{vertices}_T = \{U,A\} \cup d$
   $\mathrm{committed}_T = \{A\}$
   $\mathrm{aborted}_T = d$

   c. If $\alpha = \#\mathrm{abort}(i,j)\ A$, then $L(\alpha) = T$, where
   $\mathrm{vertices}_T = \{U,A\}$
   $\mathrm{committed}_T = \emptyset$
   $\mathrm{aborted}_T = \{A\}$

If $\alpha \in$ tloc, then $L(\alpha)$ is just the UAS at α's home node. For locations which are messages, if the message has not been sent then its location has "no information" (i.e. its UAS is the trivial UAS, T_u). If the message has been sent, then the information in the UAS for its location corresponds exactly to the information in the message, i.e. it describes what actions are known to be committed, aborted, or active as a result of the message.

<u>Event Mapping</u>

h: $\mathcal{E}_7 \rightarrow \mathcal{E}_6^*$ is defined as follows. Let *ord6* be an arbitrary total order on $\mathcal{E}_6$. For each node, i, let loc(i) be a distinguished tloc whose home is that node. (We will use this tloc to define an explicit "sender" for messages from that node. If such a tloc does not exist, then it could be created just for this purpose.)

h: create A (d) → create A $\bullet \langle\langle\{@\mathrm{create}[\beta,\alpha]\ A,d: \beta = \mathrm{creator}(A), \mathrm{home}(\alpha) = \mathrm{home}(\beta)\}; \mathit{ord6}\rangle\rangle$
commit A (d) → commit A $\bullet \langle\langle\{@\mathrm{commit}[\beta,\alpha]\ A,d: \beta = A, \mathrm{home}(\alpha) = \mathrm{home}(\beta)\}; \mathit{ord6}\rangle\rangle$
abort A → abort A $\bullet \langle\langle\{@\mathrm{abort}[\beta,\alpha]\ A: \beta = A, \mathrm{home}(\alpha) = \mathrm{home}(\beta)\}; \mathit{ord6}\rangle\rangle$

perform A,u (d) → perform A,u • <<{@commit[β,α] A,d: β = x, home(α) = home(β)}; *ord6*>>

h(**send** M) is defined as follows:

If M = #create(i,j) A,d, then h: **send** M → @create[loc(i),M] A,d
If M = #commit(i,j) A,d, then h: **send** M → @commit[loc(i),M] A,d
If M = #abort(i,j) A, then h: **send** M → @abort[loc(i),M] A

h(**receive** M) is defined as follows:

If M = #create(i,j) A,d, then h: **receive** M → <<{@create[M,α] A,d: home(α) = j}; *ord6*>>
If M = #commit(i,j) A,d, then h: **receive** M → <<{@commit[M,α] A,d: home(α) = j}; *ord6*>>
If M = #abort(i,j) A, then h: **receive** M → <<{@abort[M,α] A: home(α) = j}; *ord6*>>

We map local events to the corresponding local event at Level 6, followed by a sequence of communications events that "inform" all other locations based at the same node of the event. (Note that we use the qualifier "(d)" on local events at Level 7.) We map a send event to a communications event at Level 6 with the message slot as the destination. (The "sender" at Level 6 is an arbitrarily selected tloc at the sending node.) We map a receive event to a sequence of communications events at Level 6 with the message slot as the sender, and all tlocs at the receiving node as receivers. In general we map a single *per-node* event which affects the node's state to to a sequence of *per-location* events -- one for each tloc whose home is that node.

10.3 Proof of Possibilities Map for h_{76}

We now show that h is a possibilities map.

Lemma 10.3.1: h preserves initial states.

Proof: Let $h(D_0) = \langle L,V \rangle$; then

V = valuemap(D_0) = V_0, and
$L(\alpha) = T_u$ if $\alpha \in$ Msgs, since D_0.buf = $\emptyset$
If $\alpha \in$ tloc, then $L(\alpha) = D_0$.home(α).l = T_u.
Thus $L = L_0$. ∎

Lemma 10.3.2: h preserves transitions.

Proof: We must show that if D $\in$ PRE_7(e) $\cap$ $\mathcal{R}_7$, and h(D) $\in$ PRE_6(h(e)) $\cap$ $\mathcal{R}_6$, then h(De) = h(D)h(e).

Let De = D1, h(D) = <L,V>, h(D1) = <L1,V1>, and <L,V>h(e) = <L',V'>, then we must show that L1 = L', and V1 = V'.

We argue the cases e = create A (d), e = **send** M, and e = **receive** M for M = #create(i,j) A,d. Other cases are similar.

1. e = create A (d). Let β = creator(A), i = home(β).
   h(e) = create A • <<{@create[β,α] A,d: home(α) = i}; *ord6*>>.

   From transitions T7.1a,b, we have
   D1.buf = D.buf, D1.j = D.j $\forall$ j $\neq$ i,
   D1.i.v = D.i.v,
   #vertices_{D1}[i] = #vertices_D[i] $\cup$ {A},
   #status_{D1}[i](A) = 'active'.

   Thus V1 = V, and L1(α) = L(α) $\forall\alpha \notin$ tloc(i). If $\alpha \in$ tloc(i), then @vertices_{L1}[α] = @vertices_L[α] $\cup$ {A}, and @status_{L1}[α](A) = 'active'.

   By inspection all events in h(e) only affect locations in tloc(i); thus L'(α) = L(α) = L1(α) $\forall\alpha \notin$ tloc(i).

   Define relation $\mapsto$ on tloc(i) as follows: $\alpha 1 \mapsto \alpha 2 \Leftrightarrow \alpha 1 = \beta$, or @create[$\beta$,$\alpha$1] A,d precedes @create[$\beta$,$\alpha$2] A,d in *ord6*. ($\mapsto$ is reflexive.) Let <L2',V2'> = <L,V>u, where u is the prefix of h(e) up to and including event @create[β,α2] A,d. We can show inductively that
   @$\text{aborted}_{L2'}$[α2],
   V2'(α,A) = V(α,A) $\forall\alpha \in$ obj(i), A $\in$ act,
   $\alpha 1 \mapsto \alpha 2 \Rightarrow$ A $\in$ @$\text{active}_{L2'}$[α1],
   ~ ($\alpha 1 \mapsto \alpha 2$) $\Rightarrow$ A $\notin$ @$\text{vertices}_{L2'}$[α1].

   (We will not carry through the details of the induction here. The only subtle point is that event @create[β,α2] A,d cannot affect V(α2) (if α2 $\in$ obj(i)): If B $\in$ V(α2), then by Lemma 8.3.1a, B is live in L(α); thus B cannot be a descendant of an action in d. Note that we can apply 8.3.1a because we know <L,V> $\in$ $\mathcal{R}_6$, and I4' is invariant in L6 by Lemma 9.5.2.)

   By applying the inductive result to the total sequence h(e), we conclude that

   V' = V and L'(α) = L(α) for all α in tloc(i). Thus V1 = V', and L1 = L'.

2. e = **send** M, M = #create(i,j) A,d.
   h(e) = @create[loc(i),M] A,d.

   D1.buf = D.buf $\cup$ {M}; D1.i = D.i $\forall$i $\in$ Nodes.

   Since valuemap(D) does not depend on D.buf, V' = V. But M $\notin$ obj, so h(e) cannot affect V (in L6), $\Rightarrow$ V1 = V. Thus V1 = V'.

   Obviously L1(α) = L(α) unless α = M. But D1.i = D.i for all i $\in$ nodes, and if M' $\neq$ M, then M' $\in$ D.buf $\Leftrightarrow$ M' $\in$ D1.buf. Thus L'(α) = L(α) for all $\alpha \neq$ M.

   For α = M, L'(α) = T', where
   $\text{vertices}_{T'} = \{U,A\} \cup d$
   $\text{committed}_{T'} = \emptyset$
   $\text{aborted}_{T'} = d$

   Let L1(α) = T1, L(α) = T, then
   $\text{vertices}_{T1} = \text{vertices}_T \cup \{A\} \cup d$
   $\text{committed}_{T1} = \text{committed}_T$
   $\text{aborted}_{T1} = \text{aborted}_T \cup d$

   But if M $\in$ D.buf, then T = T' $\Rightarrow$ T1 = T'. If M $\notin$ D.buf, then T = T_u $\Rightarrow$ T1 = T'.

   Thus L1 = L'.

3. e = **receive** M, M = #create(i,j) A,d.
   h(e) = $\langle\langle$\{@create[M,α] A,d: home(α) = j\}; *ord6*$\rangle\rangle$.

   At node j, A is added to #vertices_D[j] (and made active if not already there), and d is merged into #aborted_D[j]; descendants of d are discarded from D.j.v.

   We show L1 = L' (the argument that V1 = V' is similar).
   Let $\alpha \in$ loc. If $\alpha \in$ Msgs, then clearly L'(α) = L1(α) = L(α). If home(α) $\neq$ j, then again L'(α) = L1(α) = L(α). Otherwise let L(α) = T, L1(α) = T1, L'(α) = T'. Then L(α) = D.j.l; L'(α) = D1.j.l.
   Thus $\text{vertices}_{T'} = \text{vertices}_T \cup \{A\}$, $\text{aborted}_{T'} = \text{aborted}_T \cup d$ (from transitions T7.a,c).

   But T1 differs from T by the effects of the message @create[β,α] A,d, in h(e), which has identical effect (from transitions T6.a,c), $\Rightarrow$ T1 = T'. Thus L1 = L'.

Lemma 10.3.3: h preserves preconditions.

Proof: We must show that if D $\in$ PRE_7(e) $\cap$ $\mathcal{R}_7$, and h(D) $\in$ $\mathcal{R}_6$, then h(D) $\in$ PRE_6(h(e)).

Let h(D) = ⟨L,V⟩.

We argue the cases e = perform A,u (d), e = **send** M, and e = **receive** M for M = #create(i,j) A,d. Other cases are similar.

1. e = perform A,u (d). Let x = object(A), i = home(x).
   h(e) = perform A,u • ⟨⟨{@commit[x,α] A,d: home(α) = x}; *ord6*⟩⟩.

   First we show that ⟨L,V⟩ ∈ PRE_6(perform A,u). L(x) = D.i.l, V(x,a) = D.i.v(x,a), by definition of h.

   a. A ∈ #active_D[i], by P7.4a,
      ⇒ A ∈ @active_L[x].

   b. A ∈ prop-desc(D.i.v(x).holder), by P7.4b,
      ⇒ A ∈ prop-desc(V(x).holder).

   c. u = D.i.v(x).value, by P7.4c,
      ⇒ u = V(x).value.

   d. anc(A) ∩ #aborted_D[i] = ∅, by P7.4d,
      ⇒ anc(A) ∩ @aborted_L[x] = ∅.

   Now let e' be the prefix of h(e) preceding @commit[x,α] A,d (for some α whose home is i), and let ⟨L1,V1⟩ = ⟨L,V⟩e' (in L6). We show that ⟨L1,V1⟩ ∈ PRE_6(@commit[x,α] A,d):

   a. We must show that A ∈ @committed_{L1}[x]. But event perform A,u must be in e', ⇒ A ∈ @committed_{L1}[x].

   b. We must show d = @aborted_{L1}[x]. But d = #aborted_D[i] by P7.4e, ⇒ d = @aborted_L[x].
      But none of the events in e' can change @aborted_L[x] (perform A,u obviously does not change @aborted_L[x], and if event @commit[x,x] A,d occurs in e', then d ⊆ @aborted_L[x] already). Thus d = @aborted_{L1}[x].

2. e = **receive** M, M = #create(i,j) A,d.
   h(e) = ⟨⟨{@create[M,α] A,d: home(α) = j}; *ord6*⟩⟩.

   None of the events in h(e) affect L(M), and the precondition for each event @create[M,α] A,d depends only on L(M). Thus it suffices to show that ⟨L,V⟩ ∈ PRE_6(@create[M,α] A,d) for all α.

But M ∈ D.buf by P7.6a, so L(M) = T, where
$\text{vertices}_T = \{U,A\} \cup d$,
$\text{committed}_T = \emptyset$,
$\text{aborted}_T = d$.

Thus

a. A ∈ $@\text{active}_L[M]$

b. d ∈ $@\text{aborted}_L[M]$

3. e = send M, M = #create(i,j) A,d.
h(e) = @create[loc(i),M] A,d.

home(loc(i)) = i, by definition, so L(loc(i)) = D.i.1.

a. A ∈ $\#\text{active}_D[i]$, by P7.5a,
⇒ A ∈ $@\text{active}_L[\text{loc}(i)]$.

b. d = $\#\text{aborted}_D[i]$, by P7.5b,
⇒ d = $@\text{aborted}_L[\text{loc}(i)]$. ▮

Theorem 10.3.4: h is a possibilities map.

Proof: By Lemma 10.3.1, h preserves initial states. By Lemmas 10.3.2 and 10.3.3, and Lemma 4.2.2.6, h preserves events. Thus h is a possibilities map. ▮

10.4 Mapping from Level 7 to Level 0

We can now prove the main theorem of this thesis: valid execution sequences of our lowest-level model (Level 7), when suitably interpreted, generate only view-serializable action trees.

Main Theorem: Let g: $\mathcal{S}_7 \rightarrow \mathcal{S}_0^*$ be defined by

$$g = h_{76} \circ h_{65} \circ h_{54} \circ h_{43} \circ h_{32} \circ h_{21} \circ h_{10}$$

Let v ∈ $\mathcal{V}_7$ be some valid execution sequence in L7. Then $T_i g(v)$ is a view-serializable action tree.

Proof: We have shown that each $h_{i+1,i}$ is a possibilities map (Theorems 6.4.4.1, 6.5.2.1, 7.4.5, 8.4.5, 9.4.5, 9.5.1, and 10.3.4). By Lemma 4.2.2.5, each $h_{i+1,i}$ is a valid interpretation. By repeated application of Lemma 4.1.3.2, g is a valid interpretation from L7 to L0. Thus $v \in \mathcal{V}_7 \Rightarrow g(v) \in \mathcal{V}_0$. By Lemma 6.1.1, $T_i g(v)$ is view-serializable. ∎

11. Conclusions

11.1 Summary and Evaluation

We have presented a detailed proof that a particular transaction system model satisfies our definition of internal consistency. The proof was structured on several levels, corresponding to different levels of abstraction of the transaction system. While the lowest-level model is still quite "abstract" in that it is far removed from an actual implementation, we feel that it captures many of the basic design decisions made for the Argus transaction system.

We believe our work has made two contributions: First, we have formalized internal consistency and we have related this formal condition to a particular orphan detection strategy. Second, we have explored a method for multi-level correctness proofs which might be useful in other contexts.

11.1.1 Orphan Detection and Internal Consistency

Our definition of view-serializability appears to be a useful condition for internal consistency. In the development of the Argus orphan algorithm, designers have often relied on particular scenarios where inconsistencies arose to justify the need for including certain information in messages (or writing certain information to stable storage.) While this type of reasoning can demonstrate shortcomings in the algorithm, it cannot prove the algorithm correct (we cannot "prove by example.") Perhaps the results of this thesis, and future extensions of these results, can partly subsume this "reasoning by scenario."

Although we have ignored crashes in our system models, the view-serializability condition appears to be applicable in an environment with crashes. We have applied this condition to scenarios of inconsistencies in Argus which result from crashes; these inconsistencies can be explained by showing that an action does not have a serializable view tree. (Since view-serializability is a *sufficient* condition for internal consistency, any *inconsistency* should be explicable by the absence of a serializable view tree.)

11.1.2 Algebraic Models

The multi-level structure of our correctness proof yields at least two benefits: First, since adjacent levels are generally closely related, the possibilities maps (and proofs of possibilities maps) between adjacent levels are relatively simple. Although we employ many levels, overall complexity is reduced and understandability of the mappings is enhanced.

Second, because the higher-level models are more abstract, they might prove to be useful abstractions of different implementations. At Level 1 we describe the ANC-ABORT property, at Level 2 we describe a specific orphan detection precondition, and only at Level 5 do we explain how this detection is carried out locally by piggybacking aborts lists onto messages. A different orphan strategy could be described at lower levels, but the higher-level models might still apply. As a trivial example, if all orphans are always exterminated immediately, then it is easy to show that condition ANC-ABORT from Level 1 is satisfied. Thus the correctness proof from Level 1 could be carried over to a system using immediate extermination. As another example, if we change the specific information piggybacked onto create and commit messages at Level 5 (for example, we might choose to send only a covering subset of the known aborts set) then the Level 4 model might still apply.

Our notion of "homomorphism" is unusual in that we allow "possibilities" mappings to *sets* of states at higher levels. This approach allows us to explain the "auxiliary state" technique as a particular kind of possibilities map. For our algebra hierarchy, we used a multiple-state augmention mapping between Levels 6 and 5. We speculate that the use of possibilities maps instead of auxiliary state variables might simplify some correctness proofs.

11.2 Directions for Further Research

The application of formal techniques to distributed transaction systems is a vast topic; we limit our discussion to three possible extensions of our work.

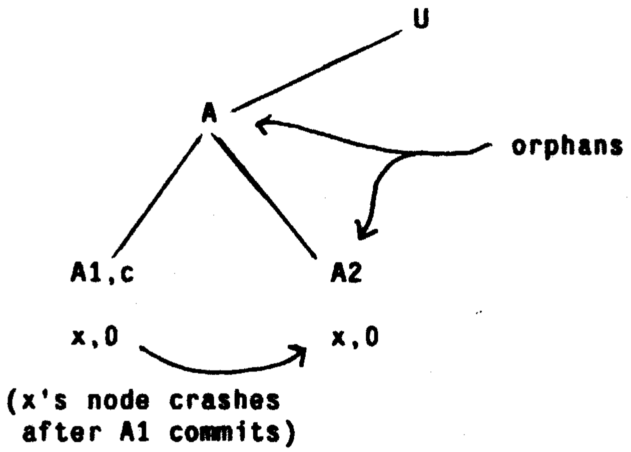
11.2.1 Crashes

The most glaring deficiency of our model is that we do not consider node crashes. Node crashes are a more difficult problem than explicit aborts because the orphans created by a node crash might be ancestors (or relatives) of actions which ran at the crashed node and committed. The "infected" ancestor can commit arbitrarily far up the action tree before the crash is discovered (though it will eventually be caught at the top level during two-phase commit if it is not caught sooner).

The (visible-data-closed) view tree which we used to prove view-serializability for the explicit aborts case will not work for a crash model. It is possible that a datastep can be "visible" to another access since it has committed to their least common ancestor, but the effect of this datastep might have been undone by a crash. Consider the tree of Fig. 11.1, for example. Object x has initial value 0. Action A spawns concurrent children A1 and A2. Action A1 runs, increments x, and commits to A. Then x's node crashes, allowing A2 to get a lock on x. Action A2 cannot see the effects of A1, because x's node crashes after A1 commits to A. A2's view is consistent, because there exists a serializable view tree for A2, but this view tree does not include A1. (A2 is an orphan, because A is an orphan, but A2 is not yet a "bad" orphan.) Note also that if A2 commits to A, then A's view becomes inconsistent. Thus an orphan detection strategy for a crash model must place restrictions on the *commit* of actions; for the explicit aborts case, we have shown that it is sufficient to put a precondition on perform events.

We speculate that a high-level notion of "depending on a crash" could be developed to parallel our notion of depending on an abort, and that a sufficient condition for view-serializability could be expressed in terms of these dependencies. Piggybacking of crash count information would appear at lower levels. A better approach would be to somehow unify aborts and crashes (i.e., treat them both as particular cases of a higher-level event), but we have made little progress in this direction.

Fig. 11.1. Consistent View of Orphan Arising from a Node Crash



11.2.2 Lower-Level Models .

Although our lowest-level model is "distributed," it ignores many of the optimizations and complications of a real orphan detection algorithm. A more satisfying "correctness proof" would extend our bottom level to even lower-level models which are closer to a real design. At least two areas for refinement may be explored: First, since the system history of aborts will grow without bound, any operational orphan algorithm will not send DONE in entirety on each message. Reducing this overhead will require some connection information or garbage collection scheme (perhaps using some variant of orphan *expiration* [Nelson81]). It would be useful to prove that these modifications are indeed optimizations in that they do not violate internal consistency.

Second, our model describes the *possible* flows of information, but it does not describe strategies for actually sending messages. (For example, do actions inform descendants immediately when they commit, or do they answer to queries from descendants?) Since our work focuses on correctness of reachable states, we have been able to ignore these questions. Of equal interest to designers, though, are properties of *liveness* (for example, will a commit message ever arrive) and bounds on delays. Formalization of these properties might require fundamentally different mechanisms.

As lower-level models become more detailed, they will approach specifications for the programs of a transaction system. At this point the boundary blurs between these correctness proofs and program verification.

11.2.3 User-Defined Atomic Data Types

We have limited the objects in our model to simple atomic objects implemented using mutual exclusion locks and a stack of versions. For some applications these objects might be inefficient: different implementations of atomic objects might provide additional concurrency or a more efficient backup and recovery mechanism. As explained in [Weihl82], the "atomicity" of a data type depends on the semantics of the operations available to users of that type. As a trivial example, if a type is "immutable" (none of the operations change the abstract object), then it is automatically atomic. Our serializability condition is insufficient to describe this more general notion of atomicity.

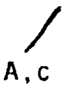
More general "user-defined" atomic types can be constructed from basic atomic objects (like those in our model) and completely *nonatomic* objects (which provide no synchronization or recovery).

(Again, see [Weihl82] for examples of these constructions.) Because the effects of aborted actions might not be undone, undetected orphans can violate *external consistency* through non-atomic data (with catastrophic effects). Thus an orphan detection strategy is more important for systems which allow non-atomic objects. Although orphan detection does not guarantee view-serializability for systems with non-atomic objects, it might guarantee weaker properties which are useful to programmers trying to use non-atomic objects to construct atomic types. We have begun to explore these properties (and more complex models which incorporate general atomic types). For example, it is relatively easy to show that the orphan detection strategy we have modeled constrains the order of datasteps on a non-atomic object to be consistent with the sequence ordering. (Without orphan detection, even this weak condition might not hold.)

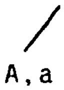
Appendix I - Notational Conventions

Fig. I.1. Conventions for Figures

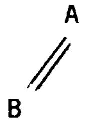
The action tree, T, is usually implicit



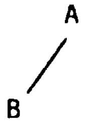
--- $A \in \text{committed}_T$



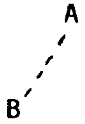
--- $A \in \text{aborted}_T$



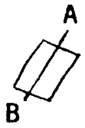
--- $A = \text{parent}(B)$



--- $A \in \text{prop-anc}(B)$



--- $A \in \text{anc}(B)$



--- $\text{prop-anc}(B) \cap \text{prop-desc}(A) \subseteq \text{committed}_T$

$A \boldsymbol{\longrightarrow} B$ --- $(A,B) \in \text{seq}_T$

$A \longrightarrow B$ --- $(A,B) \in \text{data}_T$

Fig. I.2. Cross-Reference of Invariants to Lemmas

Invariant Symbol	Lemma(s)
Ia	6.3.1.1.2, 6.3.1.1.4, 6.3.2.2, 6.3.3.1, and 6.3.3.2
Ja	6.3.1.1.1, 6.3.1.1.3, 6.3.3.3, and 6.3.3.4
Sa	6.3.1.2.1 through 6.3.1.2.14
I3	7.3.2
J3	7.3.1
I4	8.3.1
I4'	8.3.1 except for 8.3.1e
I5	9.3.3

References

[Best81] E. Best and B. Randell, "A Formal Model of Atomicity in Asynchronous Systems," Acta Informatica 16, 1981, pp. 93-124.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," CACM 19, 11, November 1976, pp. 624-633.

[Liskov82] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," 1982 Ninth Annual ACM SIGACT-SIGPLAN Symposium on PRINCIPLES OF PROGRAMMING LANGUAGES, Albuquerque, NM, January 25-27, 1982, pp. 7-19.

[Lynch82] N. Lynch, "Concurrency Control for Resilient Nested Transactions," to appear in the Proceedings of the Second ACM SIGACT-SIGMOD Symposium on PRINCIPLES OF DATABASE SYSTEMS, Atlanta, GA, March 21-23, 1983.

[Moss81] E. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing," M.I.T. Laboratory for Computer Science TR 260, April 1981.

[Nelson81] B. Nelson, "Remote Procedure Call," Carnegie-Mellon University Technical Report CMU-CS-81-119, May 1981.

[Papa79] C.H. Papadimitriou, "The Serializability of Concurrent Database Updates," JACM 26, 4, October 1979, pp. 631-653.

[Reed78] D. Reed, "Naming and Synchronization in a Decentralized Computer System," M.I.T. Laboratory for Computer Science TR 205, September 1978.

[Stark83] E. Stark, "Foundations of a Theory of Specification for Distributed Systems," Ph.D. Thesis, M.I.T. Laboratory for Computer Science, Cambridge, MA, 1983, in progress.

[Weihl82] W. Weihl and B. Liskov, "Specification and Implementation of Resilient Atomic Data Types," M.I.T. Computation Structures Group TM 223, December 1982.