

Spheres of Control: An Approach to Advanced Recovery

C. Wallace* N. Soparkar

Electrical Engineering & Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
USA
{wallace,soparkar}@eecs.umich.edu

Abstract

Recovery from failures and erroneous executions is a crucial but complicated issue for concurrently accessed data systems. Increasingly sophisticated techniques are being developed to improve performance and functionality of recovery protocols. To better understand and analyze recovery schemes, we reexamine the concept of spheres of control [Dav78], using it as a unifying framework for specifying diverse recovery models simply and precisely. We constrain sphere-of-control formulations appropriately to capture transaction-oriented recovery in both centralized and distributed environments and with different types of schedules, as well as semantics-based recovery and compensation. In addition, we discuss how the operational semantics methodology of *evolving algebras* [Gur95] can model spheres of control formally and refine them to lower levels of abstraction.

1 Introduction

The ability to recover from failures and erroneous executions is an essential feature of multiuser data systems. It also has proven to be one of the most problematic aspects of data management because of its potentially high performance cost: recovery management typically involves access to slow secondary storage during both normal and recovery processing. Complex approaches have been developed to reduce the time required for recovery and to improve performance during normal processing (*e.g.*, see [MHL⁺92, AKA⁺94]). Current research tends to rely on informal descriptions of recovery algorithms, and the efforts to prove their correctness have been limited. Consequently, they remain error-prone, difficult to understand and assess, and hence familiar only to leading experts in database recovery.

The design of a recovery management module depends on the desired levels of fault tolerance and performance and the types of operations that users perform on the data. A recovery module that tightly constrains users' actions in order to simplify recovery may be preferable. On the other hand, the benefits of allowing users greater freedom of concurrent access may outweigh the costs of complicated recovery. Certain types of operations are simple to correct in the event of a failure or error, and it may be possible to exploit the semantics of the operations to ease recovery. In some cases, it may even be advantageous for performance reasons to lower the consistency standards and allow certain errors to go unchecked. Many models of recovery management have been proposed and implemented, but without a unified method for describing such models clearly and simply, it is difficult to compare and reason about them.

The ACTA framework [Chr91] has been proposed as a solution, but it has certain inadequacies as a tool for designing and understanding complex recovery systems. It is declarative in nature, specifying each

*Supported in part by NSF grant CCR-95-04375 and ONR grant N00014-94-1-1182.

recovery method solely in terms of the conditions it imposes on the system. No mention is made of how the conditions are enforced, which we believe is a more important and difficult issue. Specifications using ACTA are necessarily on a high level of abstraction, and the gap between specification and implementation appears difficult to bridge.

We feel that the work of Bjork and Davies on *spheres of control* [Bjo73, Dav73, Dav78] can play a useful role in this regard. We see two benefits to using spheres of control in this way. First, the simplicity of spheres of control descriptions make them a useful tool for design and learning. Second, the concepts of the spheres of control framework are readily formalized, allowing for the application of verification techniques.

We discuss our work on applying the operational semantics methodology of *evolving algebras* to the spheres of control framework. This work promises to provide formal specifications of recovery methods that can be refined from a spheres of control level to a more complex implementation level. Describing advanced recovery models in terms of spheres of control provides a precise yet understandable way of reasoning about their complex behavior; moreover, the descriptions are procedural in nature. The advantage of this method over a declarative one like ACTA is that the means as well as the ends of the recovery system are represented; it is clear how users' actions are controlled to ensure recovery.

In this paper, we describe our conception of the spheres of control framework and illustrate the use of spheres of control through several examples. Section 2 introduces the notions from Bjork and Davies that we use in our framework. Section 3 presents examples of recovery models and describes them in terms of spheres of control. Section 4 shows how spheres of control can be specified in terms of evolving algebras. We conclude in Section 5 by discussing the role of spheres of control in our plans for formalizing database recovery.

2 Motivation for spheres of control

In a system where users access and update shared data, errors originating from the users and from the system can jeopardize the consistency of the data. A user program may detect an error in its computation and indicate to the system that its updates are invalid, or the system may detect an error in a user's computation and invalidate its updates. Alternatively, the system itself may be in error, resulting in the inconsistency of updates over multiple users. In all cases, the system must be able to recover from the error. To do this, it must maintain information on the effects of users' computations and the dependencies between them. It must also control users' actions to prevent them from entering an unrecoverable state. Maintaining data consistency is the responsibility of the system's *recovery manager*.

In the early 1970's, Bjork and Davies introduced the notion of spheres of control in their attempt to create a universal recovery manager: a single program that could ensure data consistency at any level desired. Spheres of control represent the dependencies that arise between users as they share a data resource. In their original conception, a recovery manager would record these dependencies in terms of spheres of control and use them later to determine the effects of a user failure.

While this work was instrumental in identifying the goals and requirements of recovery, it was an unsatisfactory solution at an implementation level. Providing a wide range of levels of data consistency in a single system proved to be impractical. First, a system allowing recovery of unlimited scope must maintain information about its entire history, even if the recovery methods actually used do not need most of this information. Second, such a system can make very few assumptions about the nature of recovery, obliging the recovery manager to identify even dependencies that are not relevant to a given application. The users of such a system would have to discriminate between the relevant and irrelevant dependencies, forcing them to make recovery design decisions they should not have to make.

Systems with more restrictive recovery schemes tailored to the given application proved to be superior in practice. The *flat transaction* model [Gra81], a restricted form of spheres of control, became the predominant paradigm for recovery. Later it became apparent that this model is too limited for many applications, and recent work has concentrated on more powerful recovery mechanisms (*e.g.*, see [Elm92]). With the introduction of various advanced transaction models, it makes sense to look at recovery from the high-level

point of view that spheres of control provide. Despite their shortcomings in terms of implementation, spheres of control provide a useful way of viewing recovery at a more abstract level.

In this section, we present our interpretations of the original concepts of Bjork and Davies. Section 2.1 addresses the problems and requirements associated with recovery, and Section 2.2 describes the fundamentals of spheres of control.

2.1 Control requirements for recovery

By updating system data, a user may introduce errors not only to the system but also to other users. For example, the actions of a certain user may be deemed erroneous after the user's updates have been performed on the database. There must be some record of what the system's state was before the updates occurred, so that they can be undone and the system can return to an acceptable state after error detection. Furthermore, any users who read the effects of the erroneous updates will themselves be in error. To prevent the propagation of errors to other users, the effects of a user's updates must be contained until they are determined to be correct. Therefore, preserving data correctness involves recording users' updates and hiding their effects until it is safe to release them.

In the model of Bjork and Davies, data in a multiuser system are accessed by *agents*¹ issuing sequences of *operations* to the system. Operations are basic actions performed upon a data item (*e.g.*, reading, writing or incrementing an item's value). An agent may delegate execution of its operations to sub-agents.

The recovery manager maintains the following conditions for each agent during its execution.

- *Agent control* guarantees the containment of an agent's actions to preserve data consistency. An agent performs a meaningful unit of work, consisting of one or more operations, which preserves data consistency if performed as a whole. If an agent is found to be in error, consistency may be maintained by undoing the agent's updates. For instance, a strong form of agent control would ensure the complete isolation of each agent by allowing it to release its results only after its successful termination. A weaker form of agent control may be achieved by allowing *active* (unterminated) agents to release their results while monitoring the agents that access these results. Agent control involves the following properties.
 - *Atomicity* ensures that the results of each agent are recorded in their entirety or not at all. It is the task of the system to ensure that the effects of an agent can be completely undone. Atomicity contains the results of an erroneous agent by ensuring that none of its results persist.
 - *Commitment* ensures that an agent which terminates successfully releases its results. Furthermore, when an agent *a* affects the processing of agent *b*, the results of *b* may be released only if permitted by *a*. This means that the scope of each agent's effects is maintained until it is considered safe to make these effects persistent. Commitment contains erroneous results by limiting the number of agents affected by them.
- *Recovery control* guarantees that the system can return to an acceptable state if one or more agents are found to be in error. Bjork and Davies distinguish the following types of recovery:
 - *Pre-commit recovery* takes place before the commitment of the erroneous agent. If the effects of the agent's operations have been effectively limited through agent control, this type of recovery can be performed simply by not releasing the agent's results.
 - *Post-commit recovery* occurs after the erroneous agent has committed. This type of recovery involves not only the erroneous agent but also any agents that have been affected by its committed updates. To perform this type of recovery, the system may have to record the operations of agents and the dependencies between them even after their commitment.

¹ We depart from the terminology of Bjork and Davies at several points. Here we use the term "agent" in place of "process".

2.2 Spheres of control

Spheres of control (or simply *spheres*) define logical boundaries around sets of operations to ensure both agent and recovery control. A sphere stores the information that the system needs to commit a agent’s updates (in the case of successful termination) or undo its effects (in the case of an error). The sphere also serves to contain its agent’s updates until they are deemed valid.

Operation requests include the following basic types. An *access* involves copying a data item from another sphere. In an *update*, an operation is performed on a data item and the result is stored in the sphere. *Committing* a agent’s update set means revealing its changes to other agents; this involves copying the changes made by the agent to its parent sphere. Finally, when a sphere is found to be in error, *recovery* uses the information in the sphere (including references to dependent spheres) to return it to a normal state.

Spheres originate from two sources:²

- Associated with each agent is an *agent-defined sphere (agt-sphere)* that encapsulates its updates. A agent’s sphere must contain the spheres of all its sub-agents, so that it can contain their effects. In addition, it must have copies of the data items the agent has accessed or modified. This allows it to reveal its updated values at commitment time.
- A *system-defined sphere (sys-sphere)* is generated and maintained by the system whenever it detects an inter-agent dependency not covered by any agt-sphere. Dependencies can arise in the following ways.
 - An agent q may access a data item updated by another agent p , forming an *access dependency*. We call p the *accessee* and q the *accessor* in such a dependency. This affects recovery, as an error in p may also cause q to be in error. To record this information for a potential future recovery, the system generates an *access sphere* enclosing the spheres of both agents.
 - An agent a may update a data item x updated by other processes, forming an *update dependency*. We call a the *updater* in such a dependency. Here the order of operations must be maintained so that if a is found to be in error, an acceptable value for x can be constructed from the previous updates of x . The system generates an *update sphere* enclosing a and the previous updaters of x .

A *schedule* is a sequence of operations performed by a data management system. As an illustration, consider the following schedule of read and write operations performed on a data item x :³

$$w_x^1 \ w_x^2 \ w_x^3 \ r_x^4$$

The sphere configuration after these actions is shown in Figure 1. Each agent has a sphere, represented by a box; sys-spheres are represented by ovals. Agent 1’s update creates an update sphere W_x^1 around S^1 . Agent 2’s update creates an update sphere W_x^2 , containing S^2 and the previous updater of x in W_x^1 . Agent 3’s update creates an update sphere W_x^3 , containing S^3 and the previous two updaters of x in W_x^2 . Finally, Agent 4’s access creates an access sphere R_x^3 , containing S^4 and the agt-sphere S^3 from which it accessed x .

If Agent 3 fails at this point, all the information necessary for recovery is present in the sphere configuration. First, a value for x can be determined by the history of updates to x , found in W_x^3 . For instance, the value of the last write to x , by S^2 , is contained in W_x^3 . Second, the agent affected by Agent 3’s update, which needs corrective action, is contained in R_x^3 . If the assumption is that readers of failed agents must abort, then S^4 can simply be removed.

²Here we use the term “agent-defined sphere” instead of “static sphere of control” and “system-defined sphere” instead of “dynamic sphere of control”.

³The execution of an operation is represented as op_x^A , where op is the operation type, A identifies the agent that executed the operation, and x identifies the data location on which the operation is performed.

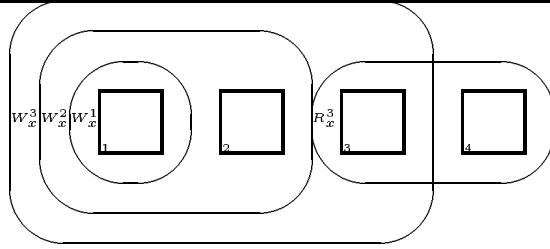


Figure 1: A sphere configuration, showing data dependencies between agents.

3 Describing transaction systems with spheres of control

In this section, we focus on specific transaction models, providing specifications that point out the details of each system. Section 3.1 focuses on systems that simplify recovery by tightly controlling agents' actions. Section 3.2 deals with more lenient recovery systems which require more effort in recovery. Section 3.3 describes how control over agents' actions can be loosened and recovery actions can be lessened by using properties of the operations' semantics.

3.1 Transactions with strict schedules

A *transaction* is a sequence of operation requests issued by a user and followed by a termination request. Requests within a transaction include *read* and *write* requests and the termination requests *commit* and *abort*. The updates of a committing transaction are released to the entire system, while those of an aborting transaction are discarded. Although agents execute concurrently, they can be prevented from reading or overwriting uncommitted values by ensuring that the schedule of requests is *strict*. A strict schedule is one in which every writer to a data item x terminates before the next read or write to x . In this way, a high level of agent control is maintained.

3.1.1 Flat transaction systems

We begin with a simple model of data consistency in which each user accesses and updates data through a single transaction. A transaction may be seen as an agt-sphere which contains its updates until its termination (commit or abort), at which point it is removed. One sphere contains all other spheres and is distinguished as the system-level sphere. This is the sphere which receives all committed updates. The sphere configuration of a flat-transaction system has a relatively simple structure. Since schedules are strict, read dependencies cannot arise between uncommitted transactions, so access spheres are never generated. Moreover, at any time there will be at most one active writer to a data item x , and therefore at most one update sphere associated with a given data item.

When a transaction reads a data item x , it adds to its access set a copy of data item x with the system-level value of x . When it writes a value v to x , it adds to its update set a copy of x with value v . In addition, an update sphere containing its sphere is added to the configuration. Its sphere is destroyed when it terminates, with its updates copied to the system level if it commits.

Consider the following schedule in a flat transaction model:

$$r_x^1 \ w_x^1(5) \ w_y^2(10) \ r_z^3 \ w_z^3(15) \ c^1 \ w_x^2(20) \ a^3$$

In Figure 2, we show the sphere configuration of the system (1) before transaction 1's commit, and (2) before and (3) after transaction 3's abort. In (1), all the updates are contained in the transactions' agt-spheres, as nothing has been released. There are update spheres around each agt-sphere to note which data locations

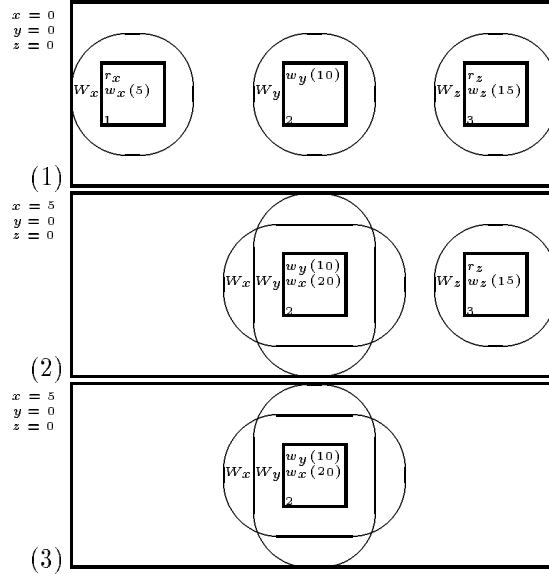


Figure 2: Strict execution of flat-transaction system.

they are updating. In (2), the agt-sphere of transaction 1 has been removed, and its updates have been released to the system-level sphere. Since transaction 2 has updated y , it is enclosed in both W_x and W_y spheres. In (3), the agt-sphere of transaction 3 has been removed, but since the transaction aborted, its updates are not released to the system.

3.1.2 Nested transaction systems

In the *nested transaction* model [Mos85], transactions may contain *sub-transactions*. Each sub-transaction has a unique parent transaction and may commit or abort without forcing its parent to do so. When a sub-transaction commits, it releases its updates only to its parent transaction, not to the entire system. A transaction may commit only after all its sub-transactions have terminated, but it may abort at any time, causing all of its sub-transactions to abort.

In a nested transaction system, let transactions 1.1 and 1.2 be sub-transactions in transaction 1, and let transactions 2.1 and 2.2 be sub-transactions in transaction 2. Consider the following schedule:

$$r_x^{1.1} w_x^{1.1}(10) w_y^{2.1}(20) w_z^{1.2}(30) c^{1.2} a^{2.2} c^{2.1}$$

In Figure 3, we show the sphere-of-control status of the system (1) before the commit of transaction 1.2, (2) before the commit of transaction 2.1, and (3) after the commit of transaction 2.1. Transaction 1.2's commit releases the update of z to transaction 1. Transaction 2.2's abort simply destroys the agt-sphere.

3.2 Transactions with recoverable schedules

As an example of recovery with a lower level of agent control, consider a system where reading or overwriting uncommitted values is permitted. A read request simply reads the value of the last uncommitted write, regardless of whether it is committed. A writer transaction then builds a set of dependents before its termination: a set of other transactions that have read the values it has written. In the case of an abort, a transaction and all its dependents must abort. Although strictness is violated in such a system, assume that the schedule is

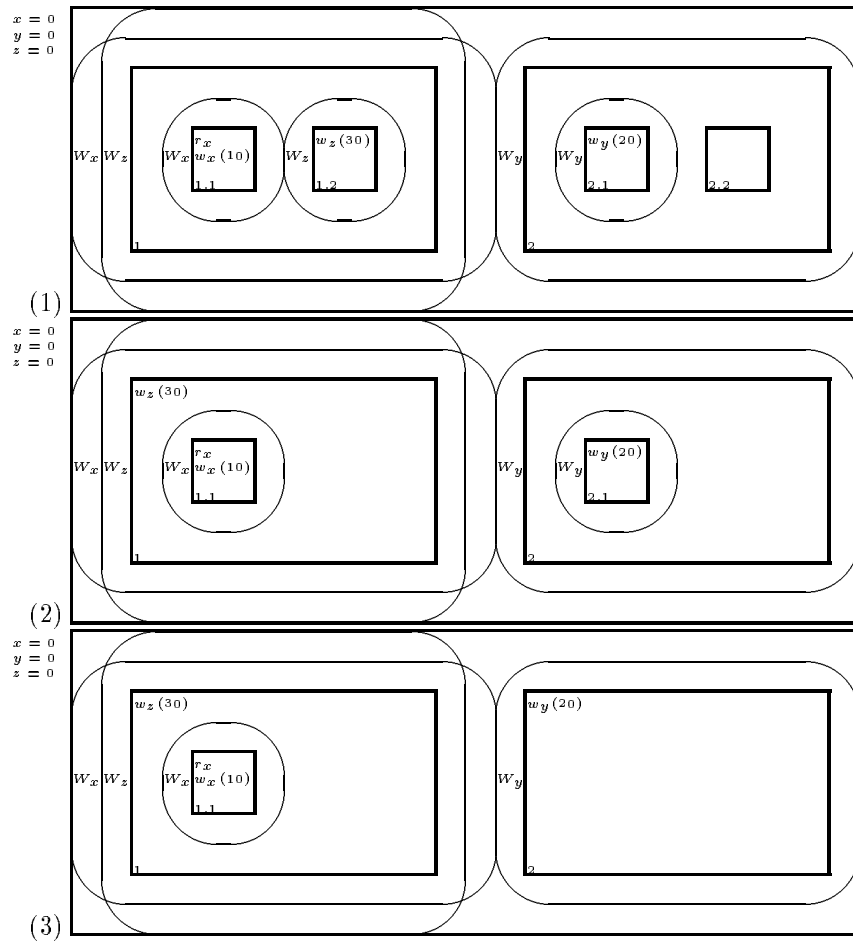


Figure 3: Strict execution of nested-transaction system.

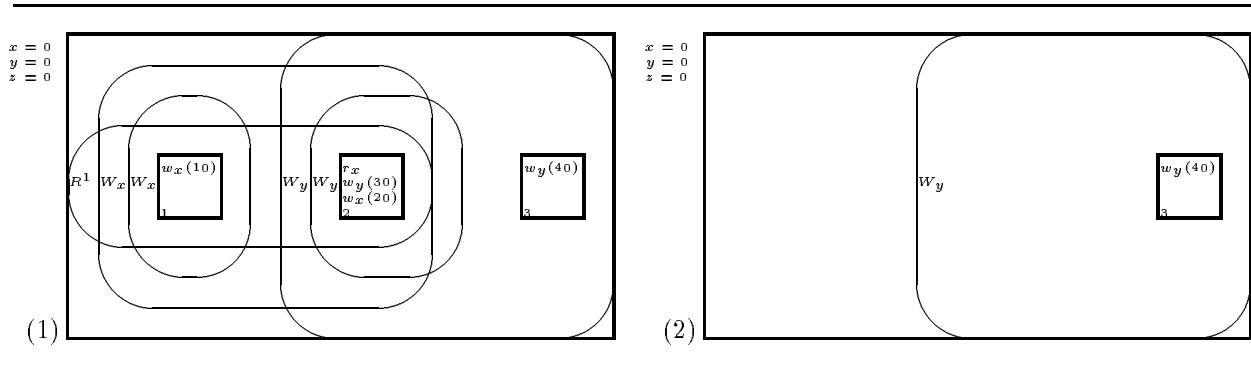


Figure 4: Recoverable execution of flat-transaction system.

recoverable: if a transaction writes a value to data item x , it must commit before any readers of this value commit.

As users' actions may generate dependencies between uncommitted agt-spheres, the system must generate sys-spheres to contain them. First, the readers of each active transaction's updates must be recorded so they may be aborted in the event of the transaction's abort. Each active transaction T_1 can be seen as having an access sphere containing the readers of its updates. When a transaction T_2 reads a value written by T_1 , it is added to T_1 's access sphere.

Furthermore, the last writer $LW(x)$ to a data item x must be recorded, as a transaction reading x will become read-dependent on $LW(x)$ and will be added to $LW(x)$'s access sphere. In addition, the previous active writers to x must be recorded, so their writes may be used in the event of $LW(x)$'s abort. From a spheres of control perspective, this can be captured by recording $LW(x)$ for each data item x and enclosing all other active writers to x in an update sphere.

Transaction requests are processed as follows. If transaction T writes a value v to data item x , a new update sphere $W(x)$ is created, enclosing T and any existing $W(x)$ spheres. If transaction T reads data item x , the x -value of $LW(x)$ is copied to T . $LW(x)$ is always found in the largest $W(x)$ sphere. If the sys-sphere $W(x)$ does not exist, there is no active writer to x , so it is the *System* value of x that is copied. Otherwise, T reads from an active transaction $LW(x)$, so it is added to the access sphere of $LW(x)$. If T commits, its updates are copied to *System*, and it is destroyed along with its access sphere and any $W(x)$ sphere immediately enclosing it. If T aborts, its access sphere and all the agt-spheres in it are also aborted, as are all $W(x)$ spheres immediately enclosing T .

Consider the following schedule in a flat transaction system:

$$w_x^1(10) \ r_x^2 \ w_x^2(20) \ w_y^2(30) \ w_y^3(40) \ a^1$$

The sphere-of-control status of the system before and after transaction 1's abort is shown in Figure 4. Transaction 2's read of x forms an access sphere around transactions 1 and 2. The history of writes to x and y are recorded through the update spheres for x and y . When transaction 1 aborts, it destroys the contents of its access sphere, namely the agt-sphere of transaction 2. The update sphere recording transaction 1's write to x is also destroyed.

3.3 Recovery exploiting special operation semantics

In a system that performs update operations other than the simple write operation considered so far, it may be possible to exploit the semantics of the operations to reduce the encapsulation required for recovery control. For example, a set of operations that commute can be executed in any order without affecting the result, so it is not necessary to record the order of such operations. In addition, if an operation has an inverse

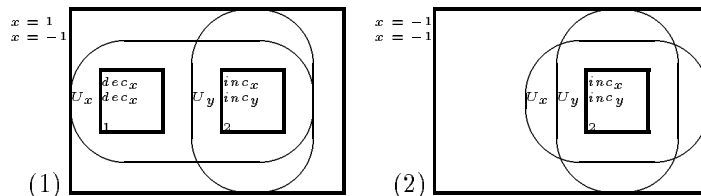


Figure 5: Execution of system with *inc* and *dec* operations only.

or *compensating* operation that undoes its effects, an instantiation of this operation may be released early and undone if necessary by executing the compensating operation.

For example, consider a system which maintains a set of numeric data items and which supports the commutative, mutually compensating operations $pred(x) = x - 1$ and $succ(x) = x + 1$. An application of $succ(x)$ can be undone by performing a compensating $pred(x)$ operation. Furthermore, it can be undone without invalidating any subsequent $succ(x)$ or $pred(x)$ operations, as the order of operations does not affect x 's value.

3.3.1 Commutativity and compensation

We start with a system in which all update operations commute and have compensating operations. In such a system, a transaction's updates need not wait for a commit but may be released immediately. The corresponding compensating operations are executed in the event of an abort. As there is no need to maintain order among the commutative operations, each data item x needs only a single update sphere $U(x)$ containing all the agt-spheres that have updated that item.

Care must still be taken when reading data items. A transaction reading a data item updated by a set of uncommitted transactions may have an invalid copy of the data item if any of the updaters abort. In this system, assume that reads of uncommitted values are simply disallowed.

Given the commutative, mutually compensating operations *inc* and *dec*, consider the following schedule:

$$inc_x^1 dec_x^2 inc_x^1 dec_y^2 a^1$$

In Figure 5, we show the sphere-of-control status of the system before and after transaction 1's abort. As updates are released immediately, the effects of the operations are immediate at the system level. The abort of transaction 1 requires that the inverses of the operations in transaction 1's sphere be released to compensate for the abort.

3.3.2 Incorporating non-commutative and non-compensatable operations

The commutativity of a set of update operations can be exploited even if it does not extend to all operations. Consider a system supporting the commutative operations *inc* and *dec* and an operation *write* which does not commute with the others. The history of updates to a data item must distinguish between occurrences of the non-commutative operator and sequences of occurrences of the commutative operators. The order of a consecutive sequence of commutative operators does not need to be maintained. Thus a new update sphere is created whenever a sequence of commutative operations on x is followed by a non-commutative operation, or vice versa. As before, reads are not permitted when there are active writers to the desired location.

Consider the following schedule:

$$inc_x^1 inc_x^2 w_x^3(10) inc_x^4 dec_x^5 c^3$$

In Figure 6, we show the sphere-of-control status of the system before and after transaction 3's commit. When transaction 3 commits, it overwrites the increment operations of transactions 1 and 2, so these operations

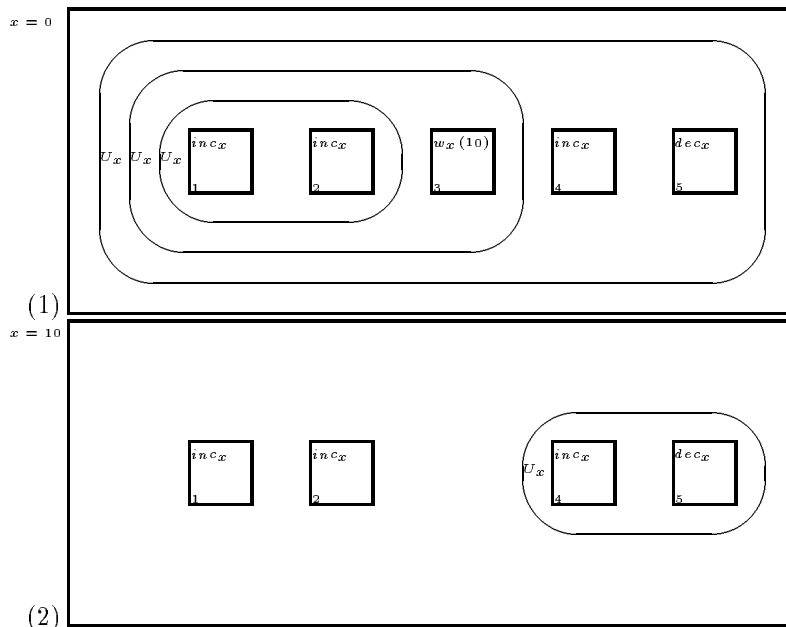


Figure 6: Execution of system with *inc*, *dec* and *write* operations.

are no longer relevant to the history of x 's updates. The update sphere containing transactions 1 and 2 is therefore destroyed.

4 Modeling spheres of control with evolving algebras

In this section, we introduce the evolving algebra concepts we need to capture spheres of control. We then illustrate the use of evolving algebras with three examples. Section 4.1 is a brief introduction to evolving algebras. Section 4.2 presents a generic spheres of control model specified in terms of an evolving algebra. The following sections refine this evolving algebra to capture the details of particular recovery models. Section 4.3 specifies the strict flat transaction approach of Section 3.1.1, and Section 4.4 specifies the recoverable flat transaction approach of Section 3.2.

4.1 Evolving algebras

This section describes the evolving algebra concepts we use in this paper; see [Gur95] for a more complete presentation of evolving algebras. A *sequential eva* (hereafter, *eva*) models a system in which an agent (in our case, the recovery manager) changes the current state in discrete steps. The behavior of the system may be seen as a sequence of states, with each non-initial state determined by its predecessor in the sequence. To model such systems, a specification method must define what a state is and how a state is obtained from its predecessor. We explain the *eva* notion of a state first, followed by the notion of state transition rules.

4.1.1 States

An *eva* state is determined by evaluating a finite collection of function names, called the *vocabulary*. For example, we model the set of updates contained in a sphere as a function: for a given sphere and location,

the function returns the value at that location in the sphere. Certain function names appear in each eva vocabulary: *true*, *false* and *undef*, the equality sign, and the names of the standard Boolean operators.

A *state* S of an eva E with vocabulary Υ consists of a nonempty set X , called the *superuniverse* of S , and an interpretation of each function name in Υ over X . The superuniverse is sorted into *universes*: in our specification, we define a universe of locations and a universe of values, among others. The name *undef* is used to represent partial functions: for any tuple outside its domain, a partial function returns *undef*.

Functions whose interpretations do not change during any execution of the eva (*e.g.*, the functions *true*, *false* and *undef*, equality, and the Boolean operators) are called *static*. On the other hand, the behavior of a system is captured by *dynamic* functions, whose interpretations do change over the course of an execution. Of these, *internal* functions change in a way determined by the state of the system. *External* functions may change in ways beyond the system's control; these represent outside forces (*e.g.*, system errors) which affect the system. We also use external functions to represent components inside the system in a high-level way. Instead of explaining the behavior of a component through possibly complex or implementation-dependent rules, we may choose to use an external function.

4.1.2 Transition Rules

Transition rules define the system dynamics that are within the control of the system; we specify the operation of the recovery manager through these rules. An *update instruction*, the simplest type of transition rule, has the form $f(\bar{x}) := v$, where f is a dynamic function name of some arity n , \bar{x} is an n -tuple of terms, and v is a term. Executing an update instruction changes the function at the given value; if \bar{a} and b are of the values of \bar{x} and v in a given state, then $f(\bar{a}) = b$ in the succeeding state.

There are three complex transition rule types.

- A *rule block* consists of a sequence of transition rules. Executing a rule block is done by executing each of its transition rules simultaneously.
- A *conditional instruction* is a transition rule of the form *if* g_0 *then* R_0 *elseif* g_1 *then* R_1 *...* *elseif* g_n *then* R_n *endif* where each *guard* g_i is a Boolean first-order term and each R_i is a transition rule. Conditional instructions operate similarly to *if...then...else...* statements in imperative programming languages. Executing a conditional instruction results in the execution of a transition rule R_i , where i is the minimal value for which g_i evaluates to *true*. If no guard evaluates to *true*, the conditional instruction performs no execution of any R_i .
- A *declaration rule* is a transition rule of the form *var* x *ranges over* U R where x is a variable and U is a universe name. Let n be the number of elements in U ; then such a rule is executed by executing n copies of R simultaneously, with x taking a different value in U in each copy.

A *run* of an eva is a sequence of *stages*, where each stage \mathcal{S} consists of a state of the eva and its number $I(\mathcal{S})$ in the sequence. For each stage \mathcal{S} after the initial stage, the interpretations of the internal functions at \mathcal{S} are obtained from the state of the previous stage by executing all enabled updates simultaneously. External function interpretations are determined arbitrarily.

4.2 High-level model

We begin with a high-level eva E_0 which captures the common components of any recovery scheme. E_0 specifies the actions that a data management system must perform, the structure of data items, and the structure of sphere configurations. From this model we may incorporate refinements to describe the details of a particular recovery model.

Every *data item* is identified by a unique *location* and contains a *value*. We define the universes *DataItem*, *Location* and *Value* accordingly. A universe *Operation* contains the operations permitted on the data items. The set of agent-defined and system-defined spheres that may arise during a run of the system is defined

```

rule Access:
if DoAccess then
  Accessor.Val(L) := Accessee.Val(L)
   $\Sigma := AddAccessSphere_{\Sigma}(Accessor)$ 
endif

rule Update:
if DoUpdate then
  Updater.Op(L) := Apply(Op, Updater.Op(L))
  Updater.Val(L) := Apply(Op, Updater.Val(L))
   $\Sigma := AddUpdateSphere_{\Sigma}(Updater)$ 
endif

rule ReleaseUpdates:
var s ranges over AgtSphere, l ranges over Location
  if ReadyToRelease(l, s) then
    Controller.Op(l) := Apply(s.Op(l), Controller.Op(l))
    Controller.Val(l) := Apply(s.Op(l), Controller.Val(l))
  endif
  if MustUndo(l, s) then
    Controller.Op(l) := Apply(Inverse(s.Op(l)), Controller.Op(l))
    Controller.Val(l) := Apply(Inverse(s.Op(l)), Controller.Val(l))
  endif
endvar

rule Terminate:
if SuccessfulTermination then
   $\Sigma := Commit_{\Sigma}(Committer)$ 
elseif UnsuccessfulTermination then
   $\Sigma := Abort_{\Sigma}(Aborter)$ 
endif

```

Figure 7: Transition rules for E_0 .

by the universe $Sphere$. We sort this universe into the universes $AgtSphere$ and $SysSphere$. $SysSphere$ is further sorted into $AccessSphere$ and $UpdateSphere$.

A sphere configuration is a partially ordered set of spheres. We define the universe $SphereCfg$ as the set of all partially ordered subsets of $Sphere$. An element σ of $SphereCfg$ consists of a domain $Dom(\sigma)$ (a subset of $Sphere$) and an order $Ord(\sigma)$ (a partial order on $Dom(\sigma)$). We use the following terminology when adding or removing spheres. When adding a sphere s within a sphere t , we use $\sigma + (s < t)$ to express the result: the partially ordered set with $\sigma \cup \{s, t\}$ as its domain and with the reflexive, transitive closure of $Ord(\sigma) \cup \{(s, t)\}$ as its order. When removing a set S of spheres from a configuration σ , we use $\sigma - S$ to represent the partially ordered set with domain $\sigma - S$ and order $\{(x, y) \in Ord(\sigma) : x \notin S \text{ and } y \notin S\}$.

The transition rules of E_0 are shown in Figure 7. All functions are dynamic unless stated otherwise. In the case of an access or update, L and Op determine the location and operation type of the action, respectively, and the Boolean-valued functions $DoAccess$ and $DoUpdate$ determine whether it is an access or an update that is to be performed. $SuccessfulTermination$ and $UnsuccessfulTermination$ determine whether an agent is terminating. The functions $Accessor$, $Updater$, $Committer$ and $Aborter$ determine the sphere for

```

rule Access:
if Req = READ and ActiveΣ(Accessor) and not ActiveΣ(WL) then
  Accessor.Val(L) := System.Val(L)
endif

rule Update:
if Req = WRITE and ActiveΣ(Updater) and not ActiveΣ(WL) then
  Updater.Op(L) := Apply(Op, Updater.Op(L))
  Updater.Val(L) := Apply(Op, Updater.Val(L))
  Σ := Σ + (Updater < WL)
endif

rule ReleaseUpdates:
var s ranges over AgtSphere, l ranges over Location
  if (Req = COMMIT) and (Committer = s) then
    System.Op(l) := Apply(s.Op(l), System.Op(l))
    System.Val(l) := Apply(s.Op(l), System.Val(l))
  endif
endvar

rule Terminate:
if Req = COMMIT then
  Σ := Σ - {Committer} - WΣCommitter
elseif Req = ABORT then
  Σ := Σ - {Aborter} - WΣAborter
endif

```

Figure 8: Transition rules for E_{strict} .

which the action is being performed. Σ represents the current sphere configuration. Given a sphere s and location l , $s.Op(l)$ and $s.Val(l)$ return the operation and value stored for l in s . For each agt-sphere s , the static function $Controller(s)$ returns the agt-sphere enclosing s (if one exists); this is the destination of s 's released updates.

In the event of a data access by a sphere s , $AddAccessSphere_\sigma(s)$ takes the configuration σ and returns the configuration with the appropriate access sphere added. $AddUpdateSphere_\sigma(s)$ is defined similarly. The functions $Commit_\sigma(s)$ and $Abort_\sigma(s)$ return the configuration given in σ with the appropriate changes required by a commit or abort of sphere s . For a location l and sphere s , $ReadyToRelease(l, s)$ determines whether a data item with location l stored in sphere s is to be released, and $MustUndo(l, s)$ determines whether the effects of the already released data item must be undone.

4.3 Flat transactions with strict schedules

E_0 may be refined as shown in Figure 8 to capture the details of a strict flat transaction system. The system-level sphere is distinguished by the static function $System$. A universe $ActionType$ contains the four types of action performed on the system, and the static functions $READ$, $WRITE$, $COMMIT$ and $ABORT$ designate each of these action types. The dynamic function Req represents the type of the current action request. The static function $Active_\sigma(s)$ determines whether sphere s is in the configuration σ .

To ensure consistency, the system must perform operations on the sphere configuration. To ensure

```

rule Access:
if Req = READ and ActiveΣ(Accessor) then
  Accessor.Val(L) := LW.Val(L)
  Σ := Σ + (Accessor < RLW)
endif

rule Update:
if Req = WRITE and ActiveΣ(Updater) then
  Updater.Op(L) := Apply(Op, Updater.Op(L))
  Updater.Val(L) := Apply(Op, Updater.Val(L))
  Σ := Σ + (Updater < WLUpdater) + (WLLW < WLUpdater)
endif

rule ReleaseUpdates:
var s ranges over AgtSphere, l ranges over Location
  if (Req = COMMIT) and (Committer = s) and not ReadDependent(Agent(NA), C) then
    System.Op(l) := Apply(s.Op(l), System.Op(l))
    System.Val(l) := Apply(s.Op(l), System.Val(l))
  endif
endvar

rule Terminate:
if Req = COMMIT and not ReadDependent(Committer, Σ) then
  Σ := Σ - {Committer} - WΣCommitter
elseif Req = ABORT then
  Σ := Σ - ContentsΣ(RAborter)
endif

```

Figure 9: Transition rules for E_{rec} .

strictness, a read from or write to a location x must only be performed if there is no active transaction that has written to x . Reading a data item does not initiate a sys-sphere, but writing to a data item does. Each update sphere is identified with a data location l . The static function W_l identifies the sys-sphere created by a write to location l , and the static function W_σ^s returns the set of update spheres in configuration σ enclosing agt-sphere s .

The system-level sphere is the source of all read accesses and the destination of all committed updates, so the functions *Accessee* and *Controller* are redefined to *System*. Accesses, updates, commits and aborts are predicated on the corresponding action type as determined by *Req*; furthermore, accesses and updates are allowed only if there are no active writers to that location, as determined by the presence of an update sphere. Finally, an agent's abort or commit removes its sphere from the configuration along with the sys-spheres it created.

4.4 Transactions with recoverable schedules

The refinements of E_0 shown in Figure 9 capture the details of a recoverable flat transaction system. First, access spheres may be generated in this system, so the static function R^s returns the access sphere containing the readers of sphere s 's updates. Second, there may be multiple active writers to a single data item, and hence multiple update spheres, so an update sphere must now be identified by the data item written and the

agt-sphere that issued the write. W_l^s returns the update sphere created by sphere s 's update at location l . The static function $LastWriter_\sigma(l)$ returns the agt-sphere s enclosed in the largest W_l^s sphere. This is the sphere from which a transaction reads l , so $Accessee$ is redefined to $LastWriter$.

When s commits, it is removed along with any immediately enclosing update spheres. W_σ^s returns the set of immediately enclosing update spheres in the given configuration. When s aborts, the contents of its access sphere (including itself) are removed. $Contents_\sigma(s)$ returns the set of spheres within s in the given configuration.

5 Discussion and Conclusions

The use of spheres of control is part of a larger project to provide a comprehensive framework for reasoning about recovery. There is a strong case for evolving algebras as a tool for specification, verification and development in the area of database recovery. Evolving algebras can be used to model an algorithm at any level of abstraction; *e.g.*, see [Gur93]. An algorithm can be specified at a high level, without concern for the implementation-specific details. These details may be added to the specification as desired through refinements of the evolving algebra. Moreover, since an evolving algebra is itself a program, automated tools can be used to implement and execute it. Thus at each level of abstraction, an evolving algebra provides not only a specification but also an executable implementation of the algorithm.

We can take our high-level spheres of control abstractions and map them to lower-level descriptions of recovery implementations. The mapping is such that the correctness properties, once proven for the high-level model, imply the correctness of the lower-level implementation. For instance, we have provided an evolving algebra specification for the simple undo/redo algorithms for transaction recovery in [WGS95]. We intend to refine our basic spheres of control models for recovery, as presented in this paper, to this lower-level implementation detail.

The flexibility of both spheres of control and evolving algebras make this a promising approach for more complex recovery problems. Evolving algebras have been used extensively in specifying distributed systems, and the spheres of control model seems readily extendable to cover distributed recovery. In addition, the expressive power of spheres of control seems well suited for complex approaches to data management such as *workflow* [GHS95, AAA⁺96].

We believe that the spheres of control approach, specified formally using evolving algebras, has the following advantages. It is procedural in nature; the actions taken by the system are described explicitly. Instead of a declarative statement of the constraints necessary for recovery, we provide a description of how the constraints may be enforced. Moreover, the description is presented at a desired level of abstraction, making it simple to study the systems. There is minimal formal machinery required, although it is based on sound mathematical foundations.

References

- [AAA⁺96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *Proceedings of the Twelfth International Conference on Data Engineering*, February 1996.
- [AKA⁺94] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Günthör, and C. Mohan. Failure handling in large scale workflow management systems. Technical report, IBM Research Division, 1994.
- [Bjo73] Lawrence Bjork. Recovery scenario for a DB/DC system. In *Proceedings of ACM 1973*, pages 142–146, 1973.
- [Chr91] P.K. Chrysanthis. *ACTA: A Framework for Modeling and Reasoning about Extended Transactions*. PhD thesis, University of Massachusetts, Amherst, 1991.

- [Dav73] Charles Davies. Recovery semantics for a DB/DC system. In *Proceedings of ACM 1973*, pages 136–141, 1973.
- [Dav78] Charles Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.
- [Elm92] Ahmed K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.
- [GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.
- [Gra81] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 144–154, September 1981.
- [Gur93] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomã, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.
- [Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
- [MHL⁺92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.
- [Mos85] J. Eliot Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.
- [WGS95] Charles Wallace, Yuri Gurevich, and Nandit Soparkar. Formalizing recovery in transaction-oriented database systems. In *Proceedings of the Seventh International Conference on Management of Data*, pages 166–185, 1995.