Masking System Crashes in Database Application Programs

Johann Christoph Freytag, Flaviu Cristian, Bo Kaehler[1]
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120-6099

Abstract

Over the last decade many techniques for recovering a consistent state for a database management system after a system crash have been proposed. However, the problem of handling system crashes in database application programs, and of masking these crashes to users of those programs, has received little attention.

This paper presents a log-based algorithm for recovering the state of database application programs after system crashes. Although the general idea of the algorithm is quite simple, the interaction between the program, the user, and the database management system has to be investigated with care. To describe the details of the algorithm clearly, we introduce a programming language with terminal input/output and database operations. By incrementally changing the semantic definition of the programming language to include operations for logging and recovery purposes, we demonstrate that the requirements for crash recovery can be met without changing the database application programs themselves.

1. Introduction

With today's computer technology, data are frequently stored and accessed by a database management system (DBMS). Besides providing a uniform interface which hides the internal representation of the data, such a system ensures the consistency of the stored data by means of concurrency control and recovery mechanisms. For instance, after a system crash, the DBMS restores the database to a consistent state [BERN83].

Usually, end users do not access data in a database by communicating directly with the DBMS. Instead, a *database application program* (DBAP) interfaces the user with the DBMS. Its purpose is to present the user with a simplified interface, to check the user's input for syntactic and semantic correctness, to prepare the response of the DBMS in a user-friendly form, etc.. One would like to make system crashes transparent to end users, so that they have the illusion of a crash-free system. Should a system crash occur, a user should not have to log on again and repeat the last inputs to the DBAP, nor should the DBAP repeat any output to the user.

The objective of this paper is to present a recovery algorithm for DBAPs that makes system crashes appear to users as delays in their interaction with DBAPs. The algorithm recovers the state of a DBAP after a crash such that users can continue their interactions with the DBAP without being aware of the occurrence of the crash. Although the general idea of our recovery algorithm is quite simple, we need to investigate the interaction between the program, the user, and the DBMS carefully. To describe the recovery algorithm in a clear and detailed form, we first define a simple programming language for writing database application programs. The language includes terminal input/output and read/write operations on the database. Using the notion of *strongest postconditions* from the area of programming language semantics [BAKK80], we first provide a precise semantics of this language when no recovery support is present. We explain our recovery technique by showing how it affects the semantics of database and terminal input/output operations.

The paper is organized as follows. In the next section we further motivate our approach to the database application recovery problem and describe the assumptions on which our solution relies. Section 3 introduces the programming language for writing DBAPs, and motivates our choice of using the concept of strongest postconditions to describe the recovery algorithm formally. Section 4 presents the recovery algorithm by defining different semantics for our programming language. Finally, Section 5 defines the general properties of our recovery algorithm, and discusses some important implementation-related aspects.

2. Motivation

As the dependence on computer services grows, the design of fault-tolerant, highly-available systems has become increasingly important over the last decade. Researchers have proposed a variety of approaches to ensure these desirable properties for computer systems. Tandem Computers and other companies

1 Author's current address: Runit, Strindveien 2, N-7034 Trondheim, Norway

offer systems with built-in hardware and software mechanisms that shield the users from different faults in hardware and software components [BART78, KATZ77]. Kim provides an overview of such systems and summarizes some of the mechanisms implemented in different systems to implement high availability in database systems [KIM84].

The concept of a *transaction* plays an important role for the recovery of databases from system crashes [BERN83, GRAY86]. The literature proposes a wide range of recovery algorithms for database management systems (DBMSs), which reinstall the effects of all those transactions that successfully finished (committed) before the system crashed, and which remove the effects of the transactions that were in progress at the time the crash occurred [BERN83].

Database application programs (DBAPs) generate transactions on the database and perform interactions with the user (see Figure 1). Their recovery from system crashes and their continued execution is the primary focus of this paper. An intuitively desirable property of recovering DBAPs is to make any crashes transparent to users working at their terminals. *Fault-tolerant systems* provide general mechanisms to implement such recovery schemes. For example, Tandem's NonStop system includes the concept of a process pair, the primary process and the backup process, together with a message-based recovery mechanism to implement fault-tolerance and availability of its computer systems [BORG83, BORR84].

In [GRAY86], Gray describes five different approaches to synchronizing the primary and the backup process. Using his classification, this paper presents an algorithm that is best characterized by the "Automatic Checkpointing" category: All messages to and from the process are saved by a message kernel [BORG84]. In case of a system crash, the backup process *replays* the messages and reaches again the state that the primary process possessed just before the crash. The message-logging kernel might also decide to save the state of the primary process, i.e. to take a checkpoint, which then can be used as a starting point for replay rather than the start state of the process. In this paper we primarily focus on the recovery by messages only, without considering checkpoints.

A message-based mechanism might be sufficient for a program that is recovered as a unit of work, independent of any other program or system component. However, since the DBAP interacts with a DBMS, we have to examine this dependency more carefully. Consider the following program T_1:

$$v \leftarrow x_{db} \; ; \; print(v) \; ; \; y_{db} \leftarrow v + 1$$

The program first reads the database variable x_{db} and prints its value on the terminal screen before assigning a new value to the database variable y_{db}. Assume that a system crash stops the execution of T_1 before the program changes the value y_{db}. Since the DBMS recovers independently of any DBAPs, it might process incoming requests in a different order than before the crash. Thus, before T_1 restarts, another program, say T_2, might have changed the value of x_{db}. Now, it is impossible for T_1 to print the same value on the terminal as before the crash and continue its execution from the program state that existed just before the crash occurred. To the best of our knowledge, none of algorithms that guarantee fault tolerance, consider this important situation.

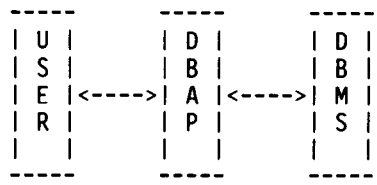

Figure 1: Relationship of the user, a DBAP, and the DBMS

It is the purpose of this paper to handle such a situation appropriately during the recovery of DBAPs.

Many of the recovery algorithms proposed in the literature are described in terms of specific system mechanisms that provide the necessary support for their implementation. Instead of relying on system-specific details, we concentrate on the *logical description* of the recovery algorithm, thus allowing its implementation on different computing systems. We use the formalism of *strongest postconditions* for the implementation-independent description [BAKK80]. Our approach is similar to the approach adopted in [CRIS85] to prove the correctness of fault-tolerant programs in the presence of system crashes and hardware fault occurrences. Our definition can serve as a specification for the implementation of a recovery component as well as as a basis for formally proving the correctness of the algorithm. However, the latter aspect is beyond the scope of this paper.

The use of strongest postconditions for the definition of the recovery algorithm achieves another desirable goal. We can clearly separate the original program from the additional run-time mechanisms which are required for recovery purposes. Our definitions show that we can add the recovery mechanisms *without* changing the original DBAP, thus making the presence of the recovery mechanism completely transparent to the program and its implementer. We confirmed this desirable property of our work by implementing the recovery algorithm as part of the Highly-Available Systems project [AGHI83], as we shall discuss in Section 5.

3. Basic Definitions

In the following subsections we define a simple programming language $\mathcal{L}$ to write DBAPs, and explain the basic concept of strongest postconditions, which is then used to define the semantics of the language.

3.1. The Programming Language for DBAPs

The programming language consists of assignment, terminal input/output, and control structure statements. For simplicity reasons, we do not include procedure and functions calls; we do not see major difficulties in the addition of such features to our language.

To access the contents of the database, we introduce a set of *database variables DB* which can be read or written by assignment statements. Reading a database variable *db* may either return

Domains for Program Variables:

DB: Set of database variables (Stable storage) *I: Terminal Input*
V: Set of program variables (Volatile storage) *O: Terminal Output*

Programming Language:

Let db ∈ DB, v ∈ V:
$\mathcal{L} ::= v \leftarrow db \mid db \leftarrow v \mid v? \mid v! \mid$ *if B then* $\mathcal{L}$ { *else* $\mathcal{L}$ } *fi* | *while B do* $\mathcal{L} \mid \mathcal{L}$; $\mathcal{L}$

Figure 2: Programming Language for DBAPs

one of the values in its respective domain or the *special* value *aborted*, by which the database management system signals an abnormal termination of the transaction to the DBAP[2]. These two operations are the only ones to access the contents of the database.

The set of (main-memory) variables of a particular program are denoted by V (volatile storage). To read input from, and write output to, the user terminal we use the notation $v?$ and $v!$, $v \in V$, respectively. Additionally, we allow sequences of assignment or input/output statements, conditional statements, and loop statements. The complete definition of the language $\mathcal{L}$ is given in Figure 2.

We allow the alternate part of the conditional statement to be omitted. The symbol B in the conditional and the loop statement denotes an arbitrary Boolean expression which evaluates (without producing side effects) to either *true* or *false*. The detailed definition of the syntax of such expressions is not important in our context, and thus is omitted. However, the Boolean expression B can only reference main-memory variables.

Notice that we also included I and O as the domains for terminal input and terminal output, respectively. We use capital letters for semantic domains (e.g. DB, V) and lower case letters for variables ranging over those domains (e.g. main memory variable v, database variable db).

For the scope of this paper we assume the *total* correctness of DBAPs, thus excluding crashes due to software faults. Furthermore, we assume that DBAPs are deterministic, that is, whenever their execution is repeated starting in the same initial state, they terminate in the same final state as before.

So far, the programming language does not include transaction capabilities. In general, a transaction is embedded into a program by a statement pair "begin_of_transaction" and "end_of_transaction". Usually, the first database operations implicitly begins a transaction. In many DBMSs, such as System R, the *commit* operation marks the end of a transaction [ASTR76]. This operation is a request to the DBMS to install the effects of all database operations of the transaction in the database. It might either succeed or fail. In the former case, the DBMS *commits* thus guaranteeing to the DBAP that the effects of all operations have been established in the database. In the latter case, the DBMS *aborts* thus signaling to the DBAP that for database-internal reasons the DBMS cannot establish the effects of transaction and that *none* of the database operations in the transaction has effected the database.

To model the transaction commit operation in our programming language, we introduce one *special* database variable *dbcom* $\in DB$ which is a *read-only* variable. When reading variable *dbcom*, it's value indicates to the DBAP whether the DBMS has committed all previous read/write operations on the database or not. Its value is *committed* if the results of all previous database operations are successfully established in the database, otherwise the value *aborted* is returned[3].

To simplify our presentation we restrict any DBAP to consist of only *one transaction*. That is, any execution of a DBAP performs a sequence of database operations (besides other operations on variables in volatile storage) before either reading the database variable *dbcom*, or reading any other database variable which returns the value *aborted* completes the transaction. Once either of these two events occurs during program execution, no additional database operations are allowed. However, the program might continue to operate on main memory variables and might perform terminal input/output operations. This restriction on DBAPs does not limit the application of our approach to program recovery, as we discuss later; it simplifies the presentation of our ideas considerably.

3.2. The Concept of Strongest Postconditions

In defining our recovery algorithm we use the concept of *strongest postcondition* from the area of programming language semantics. A detailed treatment of this subject, including the formal definition of strongest postconditions and their application, can be found in [BAKK80]. Postconditions describe the "execution effects" of statements on the program state (which consists of the current values of program variables).

2 The value *aborted* is different from any other value in any of the domains for database variables.

3 From the DBAP's point of view, it is irrelevant how the DBMS achieves the effects of a commit or an abort.

Intuitively, the strongest postcondition is the strongest (logical) statement that is valid after the execution of a program statement s if some logical precondition P is valid *before* the execution of s [BAKK80]. For example, let $v \leftarrow 1$ be a statement that assigns the value 1 to variable v. Let P be the precondition before executing the statement, and let $P[v'/v]$ denote the replacement of all free occurrences of v in P by v'. Then the strongest (logical) statement after the execution of the assignment is either $P \wedge (v = 1)$ if P does not contain v free, or $\exists v' : P[v'/v] \wedge (v = 1)$ if v occurs free in P. That is, after executing the assignment, P continues to be true except for the assertion on v whose value has been changed to 1 by the assignment. We therefore have to introduce variable v' that *replaces* v in P.

We use this concept to describe the effects of each statement, the effect of a system crash, and the "side effects" which the execution of any statement will produce for recovery purposes. In the next subsection we define the semantics without recovery support for $\mathscr{L}$ by postconditions $sp(P,s)$ with P being any precondition and s being any statement in $\mathscr{L}$. The semantics in the presence of recovery support for logging and recovery is described by postconditions denoted by $SP(P,s)$ (see Section 4).

3.3. Semantic Definition

In this subsection we define the semantics sp for all statements in $\mathscr{L}$ in the absence of recovery support. Henceforth, P denotes the precondition for any statement we discuss.

When reading the value from a database variable db into a main-memory variable v, the precondition P only changes as far as any assertion about v is concerned. Thus, we derive the strongest postcondition from P by replacing v by v' in P and by adding the assertion that v now has the value of variable db. The following statement defines the strongest postcondition formally:

$$sp(P,\ v \leftarrow db) \equiv \exists v' : P[v'/v] \wedge v = db$$

Similarly, when writing the value of a main-memory variable v into a database variable db, the strongest postcondition is derived from precondition P as follows:

$$sp(P,\ db \leftarrow v) \equiv \exists db' : P[db'/db] \wedge db = v$$

To describe the input from and output to the terminal semantically, we introduce the two variables $i \in I$ and $o \in O$. They denote a sequence of input values that the program reads during execution, and the output values generated by the program so far, respectively. For variable i, the operators $hd(i)$ and $tail(i)$ return the next user input and the rest of the input, respectively. To variable $o \in O$, we can only apply the append operator $app(o,v)$ which adds the value of variable v to the values of o already displayed on the terminal.

When reading from the terminal into a main memory variable v, we assign the value of the header of i to v and discard i's header at the same time. The postcondition derived from P includes these changes to variables v and i:

$$sp(P, v?) \equiv \exists v', i' : P[v'/v, i'/i] \wedge i = tail(i') \wedge v = hd(i')$$

Similarly, when writing to the terminal, we append the value written to the terminal to variable o that is the only variable which changes in P. Thus, we describe the effect of the output statement by the following postcondition:

$$sp(P,\ v!) \equiv \exists o' : P[o'/o] \wedge o = app(o', v)$$

We extend the definition of strongest postconditions to the three flow-control statements in $\mathscr{L}$. In the case of the sequence statement the postcondition of the first statement, $\mathscr{L}_1$, becomes the precondition for the second statement $\mathscr{L}_2$:

$$sp(P, \mathscr{L}_1 ; \mathscr{L}_2) \equiv sp(sp(P, \mathscr{L}_1), \mathscr{L}_2)$$

For the conditional statement with B as its condition, we derive the strongest postcondition by executing either the consequence with the precondition $P \wedge B$ or the alternate part with the precondition $P \wedge \neg B$. We combine both possibilities in the following definition of the strongest postcondition:

$$sp(P,\ \textit{if } B \textit{ then } \mathscr{L}_1 \textit{ else } \mathscr{L}_2) \equiv sp(P \wedge B, \mathscr{L}_1) \vee sp(P \wedge \neg B, \mathscr{L}_2)$$

Finally, we determine the strongest postcondition for the loop statement by either not executing the loop at all, which leads to the postcondition $P \wedge \neg B$ or, if B holds, by executing the body once and repeating the execution of the loop statement:

$$sp(P,\ \textit{while } B \textit{ do } \mathscr{L}) \equiv (P \wedge \neg B) \vee sp(P \wedge B, (\mathscr{L};\ \textit{while } B \textit{ do } \mathscr{L}))$$

4. The Recovery Algorithm

Based on the programming language $\mathscr{L}$ and its semantic definition for a crash-free execution without recovery support, we define the recovery algorithm for DBAPs in this section. We provide three additional semantic definitions for the programming language $\mathscr{L}$. For completeness reasons we include the crash-free semantics in this section and refer to it as the *first* semantic definition. We also refer to any program execution using the crash-free semantic definition as a *normal* execution.

With the second semantic definition we precisely describe the effects of a system crash on the execution of a DBAP, i.e. the state of the program *after* the crash. To indicate a crash during the execution of a statement, we use the crash operator ¤ of [CRIS85]. For example, ¤ $v \leftarrow db$ denotes a crash occurrence while the value of db is read into v. If a crash occurs, the program "loses" the contents of main memory and the state of the database becomes unknown to the program. Formally, let P be the precondition for a statement s. If a crash occurs during the execution of s, then the strongest postcondition after the crash is $P \setminus DB, V$. The "forget" operator "$\setminus$" applied to P removes from P all (logical) assertions about main-memory and database variables. For example, let

$$P \equiv (v = 1) \wedge (db_1 = 2) \wedge (o = ('a', 'b'))$$

be a precondition which asserts that variables v and db_1 have the values 1 and 2, respectively, and that the values a and b

were displayed on the screen. Then, the strongest postcondition after a system crash which occurs during a reading of db_1 is

$$sp(P, \natural (v \leftarrow db_1)) \equiv (o = ('a','b'))$$

that is, the state reduces to the values of variable o, i.e. the values already displayed on the terminal.

The third semantic definition describes the semantics of programs which run in the presence of a logging/recovery run-time mechanism. The definition always distinguishes between normal execution and a "replay" execution, which is the re-execution of the program after a crash.

Based on the third semantic definition, the fourth gives a precise meaning to crashes of programs which occur when logging and recovery functions are performed.

In the sequel we shall number the different semantic definitions for each language statement in $\mathscr{L}$ as follows:

1. program execution without crash
2. program execution with a crash
3. program execution including logging and recovery
4. program execution with a crash including logging and recovery

The next subsection introduces some system variables that are used for logging and recovery purposes. We then present all four semantic definitions for reading from and writing to the database, reading from and writing to the terminal, and for the restart statement that initializes the program execution after a crash. The semantic definitions for the three control statements can be found in the appendix.

4.1. System Variables for Logging and Recovery

For the description of the recovery algorithm we need to introduce several main-memory variables that keep track of different events during recovery, and two log variables $l_1, l_2 \in L$.

We define the following main-memory variables:

- $rp \in V$: is a main-memory variable that indicates if the program is being executed "normally" or if it is being "replayed" after a crash. During normal execution the variable has the value *false*. After a system crash the variable is set to *true* to replay (i.e. recover) the program up to its point of crash.
- $rbr \in V$: is called the *rollback request variable* which we use during program recovery. Its initial value is *false*. If a value is read from the database during recovery which is different from the value read before the crash, variable *rbr* is set to *true* indicating that special actions have to be taken at the end of recovery.
- $com \in V$: is a variable set during the restart of the program after a crash. Based on the values in the log, it is set to *true* if the crash occurred *after* the transaction was finished by either a transaction commit or a transaction abort. Depending on this value, the recovery algorithm proceeds differently.
- $dbabort \in DB$: is a special, write-only database variable whose initial value *true* may be set to *false* only once during the execution of the program. This operation signals an abort of the transaction to the DBMS, and forces the DBMS to remove the effects of all operations from the database. We need this operation during the recovery of DBAPs.[4]

For the log-based recovery algorithm we introduce the logs $l_1, l_2 \in L$ with operations $hd(l)$, $tail(l)$, $app(l, r)$, and $empty(l)$ to return the header record of the log l, to produce the tail of l, to append a new record r to the end of l, and to test if l is empty, respectively. We introduce two logs instead of only one for clarity reasons. Log l_1 only records values during normal execution. Since the program is deterministic we do not need to record any variable names. During restart, the contents of l_1 is copied into l_2, which then is "consumed" during the recovery. Notice that the two logs, though conceptually distinct, need not be implemented this way. We might, for example, implement them as two separate scans on the same log file.

4.2. Reading from the Database

When accessing a database variable we need to distinguish between reading the variable *dbcom* which finishes the transaction, from reading all other database variables. In Figure 3 we show the four different semantics for the latter; Figure 4 shows the semantic descriptions for the database variable *dbcom*.

During a normal execution, the main-memory variable v is assigned the value of the variable *db*. The precondition P is still true after the assignment statement, except for any assertions about v. We therefore "modify" P by substituting variable v' for v. The strongest postcondition in case of a crash is described by the second item of Figure 3. The strongest postcondition after the crash consists of precondition P with all references to database or main-memory variables removed.

The third item of Figure 3 describes the necessary changes to include logging and recovery for reading the database variable *db* into v. Instead of changing the semantics we prefer to modify the original statement to add operations for logging and recovery. C_1 describes the case of a normal execution, i.e. the replay variable *rp* is *false*. Besides assigning the value of *db* to variable v, we also log its value.

C_2 defines the changed program for a recovery after a crash. If the recovery log l_2 is not empty, we read the log to provide a value for the main-memory variable v. Furthermore, we have to test whether the database variable *db* still has the same value as before the crash. If not, the event is recorded by setting variable *rbr* to *true*, indicating that the program cannot recover into the state when the crash occurred. Special actions are then necessary to signal this event to the program at the end of recovery. We perform this test only if no changes of database variables have been detected (i.e. $rbr = false$) so far, and only if the crash

4 Usually, this operation is also available for DBAPs. For simplicity reasons, we exclude its general use in DBAPs.

1. $sp(P, v \leftarrow db) \equiv \exists v' : P[v'/v] \wedge v = db$

2. $sp(P, \sqcap v \leftarrow db) \equiv P \setminus DB, V$

3. $SP(P, v \leftarrow db) \equiv sp(\neg rp \wedge P, C_1) \vee sp(rp \wedge P, C_2)$ *where*

$C_1 \equiv v \leftarrow db;\ l_1 \leftarrow app(l_1, db);$

$C_2 \equiv$ *if* $\neg empty(l_2)$ *then*
$v \leftarrow hd(l_2);\ l_2 \leftarrow tl(l_2);$
if $((rbr = false) \wedge (com = false))$ *then* $v' \leftarrow db;$ *if* $(v \neq v')$ *then* $rbr \leftarrow true$ *fi fi*
else
$rp \leftarrow false;$
if $(rbr = true)$ *then* $v \leftarrow$ *'aborted'*; $dbabort \leftarrow true;\ l_1 \leftarrow app(l_1, 'aborted')$
else $v \leftarrow db;\ l_1 \leftarrow app(l_1, db)$ *fi*
fi

4. $SP(P, \sqcap v \leftarrow db) \equiv sp(P \wedge \neg rp, \sqcap C_1) \vee sp(P \wedge rp, \sqcap C_2)$

Figure 3: Semantic Definitions for reading any database variable, except dbcom

occurred before the end of the transaction (i.e. *com = false*). During restart we initialize both variable, *rbr* and *com* appropriately.

If the recovery log l_2 is empty, we have reached the end of recovery, thus setting the replay variable *rp* to *false*. If any difference between the current database values and the logged values has been detected during the recovery (i.e. *rbr = true*), we signal an abort to the program by returning the value *aborted*, set the *dbabort* variable to *false* to force the DBMS to abort the transaction, and record the event on the log l_1. By returning the *abort* value to the DBAP, we use the error-reporting facility provided by the transaction concept, thus avoiding additional exception handling in case the program cannot be recovered into the same state as before the crash. As the DBAP cannot distinguish the reasons for the abort, we keep the recovery mechanisms transparent to the DBAP.

The fourth item of Figure 3 defines the semantics of recoverably reading a database variable in the presence of a crash, by simply referring to the program C_1 and C_2 in the previous item. The semantic definition for a sequence of statements in case of a crash can be found in the appendix. For normal execution (case 1), the semantic description for reading database variable *dbcom*

1. $sp(P, v \leftarrow dbcom) \equiv \exists v' : (dbcom = 'committed' \wedge (P[v'/v] \wedge v = dbcom)) \vee$

$(dbcom = 'aborted' \wedge (P[v'/v] \setminus DB, V) \wedge v = dbcom)$

2. $sp(P, \sqcap v \leftarrow dbcom) \equiv P \setminus DB, V$

3. $SP(P, v \leftarrow dbcom) \equiv (\neg rp \wedge R_1) \vee (rp \wedge R_2)$ *where*

$R_1 \equiv \exists l'_1, v' : P[l'_1/l_1, v'/v] \wedge l_1 = app(l'_1, dbcom) \wedge v = dbcom$

$R_2 \equiv \exists l'_2, v' : P[l'_2/l_2, v'/v] \wedge ((\neg empty(l'_2) \wedge S_1) \vee (empty(l'_2) \wedge S_2 \wedge \neg rp))$ *where*

$S_1 \equiv v = hd(l'_2) \wedge l_2 = tail(l'_2)$

$S_2 \equiv (\neg rbr \wedge R_1) \vee (rbr \wedge dbcom = 'aborted' \wedge dbabort = true \wedge l_1 = app(l'_1, 'aborted'))$

4. $SP(P, \sqcap v \leftarrow dbcom) \equiv \{SP(P, v \leftarrow dbcom) \vee P\} \setminus DB, V$

Figure 4: Semantic Definitions for reading the database variable dbcom

is different from reading all other database variables (see Figure 4, Item 1). If the transaction commits, i.e. reading variable *dbcom* returns the value *committed*, the DBMS guarantees the DBPA that the effect of the transaction's operations have been established in the database. However, if the transaction aborts, none of the transaction's operations has effected the contents of the database. In fact, the DBAP then does not know at all what the contents of the database is. Therefore, assertions about any database variables have to be removed from the precondition. In case of a crash, the semantic description for reading database variable *dbcom* remains unchanged. Item 3 of Figure 4 includes logging and recovery for handling crashes. During normal execution, the strongest postcondition R_1 is derived from the precondition P with the additional information that variable v has been assigned the value of *dbcom* and that the value has been recorded on the log.

By the definition of the strongest postcondition R_1, the value of *dbcom* is *guaranteed* to be recorded on the log l_1. Combining the reading and recording of the *committed* value as one atomic action is important for the correctness of the algorithm. Suppose the transaction commits and a crash occurs *before* the *committed* value is recorded on the log. We then might repeat all operations of the transaction successfully and commit the transaction a second time, thus performing the operations on the database twice. We shall discuss the impact of this important requirement on the the implementation of the recovery algorithm in Section 5.

R_2 defines the postcondition for the recovery case. If the recovery log l_2 is not empty, the transaction was completed before the crash; the recorded value for variable *dbcom* is read from the log and assigned to v. If the log l_2 is empty, we test the variable *rbr* to determine if any differences between the values in the log and the database have occurred previously. However, we do not compare the value from the log with the current value from the database. Either the transaction was completed *and* the return value of variable *dbcom* was recorded or neither of the two events happened (see the definition of R_1 of Item 3). In the former case, the log l_2 is not empty and the value of the commit operation is restored. In the latter case, we proceed by either performing the commit operation or by aborting the transaction, depending on the value of variable *rbr*.

4.3. Writing to the Database

No additional operations are included for logging and recovery to write the value of a main-memory variable into the database. The semantics for all four cases are defined in Figure 5.

1. $sp(P, db \leftarrow v) \equiv \exists db' : P[db'/db] \wedge db = v$
2. $sp(P, \square\, db \leftarrow v) \equiv P \setminus DB, V$
3. $SP(P, db \leftarrow v) \equiv sp(P, db \leftarrow v)$
4. $SP(P, \square\, db \leftarrow v) \equiv P \setminus DB, V$

Figure 5: Writing a database variable

If a crash occurs the state of the database becomes unknown and main-memory is lost. Thus, the strongest postconditions are derived from P by deleting all (logical) assertions for database and main memory variables.

4.4. Terminal Input/Output

Figures 6 and 7 summarize the four different semantics for terminal input/output operations. We model the input from the terminal by reading variable $i \in I$, which contains the list of all inputs the user will provide during the execution of the program. The header of the list, which is the next input value is assigned to v. At the same time, the value is removed from the input. Similarly, in case of a terminal output operation, the displayed value is appended to the the output list o. A system crash reduces the program state to an assertion on variables i and o, that is the values which have been received form the terminal and which have been displayed on the terminal so far, respectively. Any assertions about database or main memory variables are removed from the preconditions (see Item 2 of Figure 6 and 7).

Item 3 of Figure 6 defines the logging and recovery semantics for terminal input during normal execution. Notice that reading

1. $sp(P, v?) \equiv \exists v', i' : P[v'/v, i'/i] \wedge i = tail(i') \wedge v = hd(i')$
2. $sp(P, \square\, v?) \equiv \{\exists i' : P[i'/i] \wedge (i = tail(i') \vee i = i')\} \setminus DB, V$
3. $SP(P, v?) \equiv (\neg rp \wedge R_1) \vee (rp \wedge R_2)$ *where*

   $R_1 \equiv \exists l'_1, v', i' : P[l'_1/l_1, v'/v, i'/i] \wedge (l_1 = app(l'_1, hd(i')) \wedge i = tail(i') \wedge v = hd(i'))$

   $R_2 \equiv \exists l'_2, v' : P[l'_2/l_2, v'/v] \wedge ((\neg empty(l'_2) \wedge v = hd(l'_2) \wedge l_2 = tail(l'_2)) \vee (empty(l'_2) \wedge R_1))$
4. $SP(P, \square\, v?) \equiv \{SP(P, v?) \vee P\} \setminus DB, V$

Figure 6: Semantic Definition for Terminal Input

1. $sp(P, v!) \equiv \exists o' : P[o'/o] \wedge o = app(o', v)$

2. $sp(P, \neg v!) \equiv \{\exists o' : P[o'/o] \wedge (o = app(o', v) \vee o = o')\} \setminus DB, V$

3. $SP(P, v!) \equiv (\neg rp \wedge R_1) \vee (rp \wedge R_2)$ *where*

$R_1 \equiv \exists l'_1, v', o' : P[l'_1/l_1, v'/v, o'/o] \wedge (l_1 = app(l'_1, v) \wedge o = app(o', v))$

$R_2 \equiv \exists l'_2 : P[l'_2/l_2] \wedge ((\neg empty(l'_2) \wedge l_2 = tail(l'_2)) \vee (empty(l'_2) \wedge R_1))$

4. $SP(P, \neg v!) \equiv \{SP(P, v!) \vee P\} \setminus DB, V$

Figure 7: Semantic Definition for Terminal Output

from the terminal and logging the value read is again *one atomic action*. If a crash occurs, either no input was read, or if the input was read then the value was recorded on the log. The atomicity ensures that the user does not supply the same input twice.

R_2 of Item 3 defines the semantics for the recovery case. If the log l_2 is not empty, the first value of the log is assigned to variable *v*, otherwise we read a new input value and record the value in the log l_1 at the same time. However, we *cannot* reset the replay variable *rp* to signal the end of the recovery. We might be in the middle of the transaction, thus making it necessary to test variable *rbr* during the read of a database variable and possibly to abort the transaction. We can reset variable *rp* only while reading a database variable which includes the test of variable *rbr*.

The semantics for terminal output are similar to the ones for terminal input. Instead of reading from input *i*, we now write to variable $o \in O$ which "accumulates" all output values written to the terminal so far. Notice that for logging and recovery, the operations of writing and logging the value are an atomic action to ensure that output values do not appear twice on the terminal. For terminal output, it is not necessary to record the exact value; it suffices to write one bit that indicates that an output on the terminal occurred. During the recovery, we simply suppress the output.

4.5. The Restart Statement

The restart statement initializes the program execution for recovery after a crash. It's semantic is defined in Figure 8. Since the statement is used for recovery purposes only, its semantic is only defined for cases 3 and 4. We set the replay variable *rp* to *true*, which indicates to the recovery algorithm to replay the program, initialize variable *rbr* to *false*, initialize the recovery log l_2 with the contents of l_1, check whether the transaction was completed before the crash or not, and provide a new database state P_{DB}, i.e. new values to all database variables accessed by the program. The latter is necessary since *P*, the postcondition for the restart statement, does not include assertions about any database variables.

5. Discussion

In this section we discuss two important aspects of the recovery algorithm: its properties and its implementation. In particular, we examine whether we can express the properties of the recovery algorithm more formally. Then, we briefly describe the implementation of the algorithm as part of the Highly-Available Systems (HAS) project at the IBM Almaden Research Center [AGHI83].

1. *not applicable*

2. *not applicable*

3. $SP(P, restart) \equiv P \wedge (rp = true) \wedge (rbr = false) \wedge (l_2 = l_1) \wedge (com = S_1) \wedge P_{DB}$ *where*

$S_1 \equiv \exists e \in l_1 : (e = 'committed') \vee (e = 'aborted')$

4. $SP(P, \neg restart) \equiv P \vee SP(P, restart) \setminus DB, V$

Figure 8: Semantic Definitions for the Restart Statement

5.1. Algorithm Properties

In the previous sections we concentrated on the definition of the recovery algorithm. In Section 2 we gave an intuitive description of the algorithm's properties. This subsection describes possible properties of the recovery more formally.

First and foremost, the recovery algorithm should not alter the execution of the program. Based on our notation using strongest postconditions, the extended semantic definition SP which includes logging and recovery should lead to the same final state as the original semantic sp when starting the program execution in the same initial state P. Formally, let R denote the set $\{l_1, l_2, dbabort, rbr, rp, v', com\}$ which are all the variables used for the recovery. Then $SP(P, \mathscr{L}) \setminus R = sp(P, \mathscr{L})$ should hold for all programs $\mathscr{L}$.

Another important property for the recovery algorithm should be to mask a crash from the user at the terminal. This requirement implies that the user sees a crash-free execution of the program starting in some initial state P'. If the program starts in some initial state P, crashes, recovers and successfully completes, its execution should be equivalent to one crash-free execution starting in some state P'. Let $Q \equiv (\ \square\ \mathscr{L}\ )$; *restart*; $\mathscr{L}$ be the program which crashes, restarts, and then finishes successfully. Then we can describe our requirement by the following formal statement:

$$SP(P, Q) \setminus R = SP(P', \mathscr{L}) \setminus R$$

for some P'. The initial state P' does not have to be the same as the state P. However, variable $i \in I$, which describes the terminal input, must be the same for both states.

We might strengthen our requirement by demanding that the recovery algorithm tolerates multiple crashes. We use the Kleene star * of regular expressions to express the indefinite, but finite number of crashes before the program finishes in a crash-free execution:

$$SP(P, Q) \setminus R = SP(P', \mathscr{L}) \setminus R$$

with $Q \equiv (\ \square\ \mathscr{L}\ ;\ (\square\ (restart\ ;\ \mathscr{L}))^*\ ;\ restart;\ \mathscr{L})$ for some P'.

The strongest postcondition semantic specification given in the previous section can be used to prove the correctness of our recovery algorithm with respect to the requirements stated above. However, such a proof is beyond the scope of this paper.

5.2. Implementation

Based on the specification of the previous section we implemented a recovery component for DBAPs as part of the HAS project at the IBM Almaden Research Center using the DBMS SQL/DS and the operating system VM as our experimental implementation environment. The recovery component consists of three major sub-components as shown in Figure 9. The values of each input/output operation to and from the terminal and the database are intercepted by one of the recovery components to perform necessary operations according to the specification of the algorithm in the previous section. We implemented a special log component independent of VM's file system to guarantee the

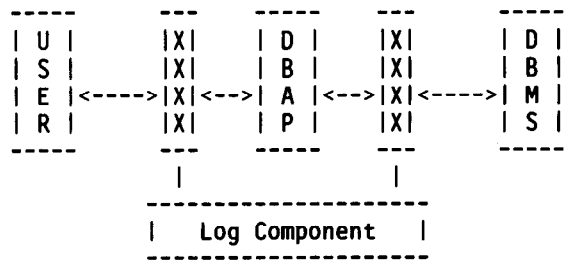

Figure 9: The User, the DBAP, the DBMS, and the Recovery Component

atomic writes of the log records. The addition of the recovery component is transparent to the DBAPs and to the DBMS.

The correctness of our recovery mechanism was based on the assumption that we can execute the input/output operations and the database commit operation atomically with the corresponding log operations. To guarantee the atomicity, we introduced internal transaction identifiers, which are stored in the log and in the database at the beginning of each transaction. If we can retrieve the transaction identifier during recovery, we know that the DBMS committed the transaction. The disadvantage of our solution is that read-only transactions always become read-write transactions.

We did not attempt to find a solution that guarantees the atomicity of terminal/input output operations with the log operations, since we believe that a complete solution to this problem does not exist. By reducing the time between any input/output operation and the corresponding log operation as much as possible, we minimized the risk of a crash between a terminal input/output and a log operation. More complicated solutions, such as the use of PCs as intelligent terminals, could further reduce the probability of losing any terminal input/output.

The specification of the recovery algorithm restricts DBAPs to consist of only one transaction. We removed this restriction for the implementation easily. For each transaction, we simply determine if it completed or not by "looking ahead" on the log. No additional changes were necessary to extend the algorithm to multiple transactions.

To improve the recovery for long programs, we extended the recovery component by a *checkpointing facility*. Periodically, the component saves a complete program state during normal program execution. In case of a crash, the log-based recovery begins with the state saved by the checkpointing component, instead of starting execution from the initial state of the program. The checkpointing component determines independently of the program which data to save in order to resume execution after a crash. However, the implementer of the DBAP had to include an explicit call for a checkpoint, thus making the recovery visible in the DBAP.

6. Conclusion

This paper describes a log-based recovery algorithm for database application programs by using the notion of strongest

postconditions borrowed from the field of programming language semantics. This formalism allows us to give a clear and detailed definition of the recovery algorithm without relying on any system-specific mechanisms. We therefore could use the high-level description as a specification for different implementations of the algorithm. Our specification also shows that we can add operations for program recovery transparently to the DBAP. The recovery algorithm presented does not require the introduction of new "error exits" in case we cannot recover the program into the state just before the crash occurred. The algorithm simply uses the already existing error exit provided by the database operations commit and read.

Additionally, we discussed more formally the properties of the recovery algorithms for DBAPs, and shortly outlined the implementation of the algorithm as part of the HAS project at the IBM Almaden Research Center.

Bibliography

[AGHI83] Aghili, H. et al., *A Prototype for a Highly Available Database System,* IBM Research Report RJ 3755, San Jose (January 1983).

[ASTR76] Astrahan, M. et al., *SYSTEM R: Relational Approach to Database Management,* ACM Transactions of Database Systems **1,2** (June 1976) pp. 97–137.

[BAKK80] de Bakker, J., *Mathematical Theory of Program Correctness,* Prentice-Hall International Series in Computer Science (1980).

[BART78] Bartlett, J., *A Nonstop Operating System,* Proceedings of the International Conference on System Sciences, Honolulu, Hawaii (January 1978).

[BERN83] Bernstein, P.A., Goodman, N., Hadzilacos, V., *Recovery Algorithms for Database Systems,* Proceedings of IFIP (1983) pp. 799–807.

[BORG83] Borg, A., Baumbach, J., Glazer,S., *A Message System Supporting Fault Tolerance,* Proceedings of the Ninth ACM Symposium of Operating System Principles, Bretton Woods, N.H. (OS System Review 17,5) (October 1983) pp. 90–99.

[BORR84] Borr, A., *Robustness to Crash in a Distributed Database: A non Shared-Memory Multi-Processor Approach,* Proceedings of the 10th VLDB, Singapore (August 1984) pp. 445–453.

[CRIS85] Cristian, F., *A Rigorous Approach to Fault-Tolerant Programming,* IEEE Transactions on Software Engineering **SE-11, No. 1** (1985) pp. 23–31.

[GRAY86] Gray, J., *Why do Computers stop and What can be done about it?,* Fifth ACM/IEEE Symposium on Reliability in Distributed Software and Database Systems, Los Angeles (January 1986) pp. 3–12.

[KATZ77] Katzman, J.A., *System Architecture for NonStop Computing,* Proceedings of the CompCon, IEEE Computer Society (February 1977) pp. 77–80.

[KIM84] Kim, W., *Highly Available Systems for Data Base Applications,* ACM Computing Surveys **16,1** (March 1984) pp. 97–137.

APPENDIX
The Semantic Definitions for the Control Statements

The Sequence Statement

1. $sp(P, \mathscr{L}_1 ; \mathscr{L}_2) \equiv sp(sp(P, \mathscr{L}_1), \mathscr{L}_2)$
2. $sp(P, \natural(\mathscr{L}_1 ; \mathscr{L}_2)) \equiv sp(P, \natural(\mathscr{L}_1)) \vee sp(sp(P, \mathscr{L}_1), \natural \mathscr{L}_2)$
3. $SP(P, \mathscr{L}_1 ; \mathscr{L}_2) \equiv SP(SP(P, \mathscr{L}_1), \mathscr{L}_2)$
4. $SP(P, \natural(\mathscr{L}_1 ; \mathscr{L}_2)) \equiv SP(P, \natural \mathscr{L}_1) \vee SP(SP(P, \mathscr{L}_1), \natural \mathscr{L}_2)$

The Conditional Statement

1. $sp(P, \mathit{if}\, B\, \mathit{then} \mathscr{L}_1 \mathit{else} \mathscr{L}_2) \equiv sp(P \wedge B, \mathscr{L}_1) \vee sp(P \wedge \neg B, \mathscr{L}_2)$
2. $sp(P, \natural\, \mathit{if}\, B\, \mathit{then} \mathscr{L}_1 \mathit{else} \mathscr{L}_2) \equiv sp(P \wedge B, \natural \mathscr{L}_1) \vee sp(P \wedge \neg B, \natural \mathscr{L}_2)$
3. $SP(P, \mathit{if}\, B\, \mathit{then} \mathscr{L}_1 \mathit{else} \mathscr{L}_2) \equiv SP(P \wedge B, \mathscr{L}_1) \vee SP(P \wedge \neg B, \mathscr{L}_2)$
4. $SP(P, \natural\, \mathit{if}\, B\, \mathit{then} \mathscr{L}_1 \mathit{else} \mathscr{L}_2) \equiv SP(P \wedge B, \natural \mathscr{L}_1) \vee SP(P \wedge \neg B, \natural \mathscr{L}_2)$

The Loop Statement

1. $sp(P, \mathit{while}\, B\, \mathit{do} \mathscr{L}) \equiv sp(P \wedge B, (\mathscr{L}; \mathit{while}\, B\, \mathit{do} \mathscr{L})) \vee (P \wedge \neg B)$
2. $sp(P, \natural\, \mathit{while}\, B\, \mathit{do} \mathscr{L}) \equiv sp(P \wedge B, \natural(\mathscr{L}; \mathit{while}\, B\, \mathit{do} \mathscr{L})) \vee \{(P \wedge \neg B)\} \setminus DB, V$
3. $SP(P, \mathit{while}\, B\, \mathit{do} \mathscr{L}) \equiv SP(P \wedge B, (\mathscr{L}; \mathit{while}\, B\, \mathit{do} \mathscr{L})) \vee (P \wedge \neg B)$
4. $SP(P, \natural\, \mathit{while}\, B\, \mathit{do} \mathscr{L}) \equiv SP(P \wedge B, \natural(\mathscr{L}; \mathit{while}\, B\, \mathit{do} \mathscr{L})) \vee \{(P \wedge \neg B)\} \setminus DB, V$