The dual-level validation concurrency control method

Zhixue Wu, Ken Moody, Jean Bacon

June 1993
The Dual-Level Validation Concurrency Control Method

Zhixue Wu, Ken Moody and Jean Bacon
University of Cambridge Computer Laboratory, UK

Abstract

Atomic data types permit maximum concurrency among transactions by exploiting the semantics of object operations. Concurrency control is needed to ensure both object level atomicity and transaction level atomicity. It must be possible to regard each operation on an object as elementary. Recovery methods for transactions which are based on atomic objects must take into account that partial results of a transaction might be seen by other transactions.

This paper presents, formalises and verifies a protocol called the dual-level validation method which can be used to provide atomicity for atomic data types. It is optimistic and has a number of advantages over previous methods. It permits maximum concurrency at the low level by allowing non-conflicting operations to be scheduled concurrently. It allows applications to cope with very large objects by supporting multi-granularity shadowing. Transaction recovery is simple to implement. The method performs well, particularly when different transactions are unlikely to access the same (sub)object simultaneously. Finally, it is well suited to a distributed environment since validation and commit are not implemented atomically.

1 Introduction

Transactions are designed to cope with concurrent execution and failures. They are therefore useful for managing computations in distributed systems in general as well as in database systems. One way of ensuring atomicity of transactions is to implement applications in terms of atomic data types: data types whose objects, atomic data objects, provide serialisability and recoverability for the transactions which use them. Atomicity of transactions is guaranteed when all objects shared by transactions are atomic objects[WL85].

If the semantics of object operations are taken into account, more concurrency can be achieved than with read-write semantics. For example, two transactions that perform a credit operation on a bank account object can proceed concurrently because credit operations are commutative. A low-level synchronisation mechanism is then required to take care of possible conflicts. For example, the credit operation is implemented as a read and a write so two credits executed concurrently could interfere with each other at the low-level. An implementation of atomic data types based on operation semantics must therefore deal with this process concurrency as well as transaction concurrency. Process concurrency is about making the object operations elementary and can be achieved by classic methods such as taking out an exclusive lock on the object for the duration of a
critical read-modify-write sequence. Our approach is, in contrast, optimistic. Transaction concurrency control is discussed below.

In an implementation of transactions based on atomic data types non-conflicting operations can be scheduled concurrently; that is, the execution schedule is inherently non-strict, i.e. the isolation property of the transaction is not supported by the implementation. Nevertheless, the atomic data objects must be recoverable to allow for failures and transaction aborts. For example, consider an object A that initially has the value £2000. If two concurrent transactions $T_1$ and $T_2$ both invoke a credit(£1000) operation on A then, providing both transactions commit, the result is that A has the value £4000. In an attempt to make A recoverable, suppose that each transaction records the value of A prior to its invocation. Consider the following sequences of events. Transaction $T_1$ changes the value of A to £3000 and records the old value as £2000. Then transaction $T_2$ sets the value of A to £4000, recording the old value as £3000. $T_2$ then commits producing £4000 as the final value for A. $T_1$ then aborts. Clearly, $T_1$ should not restore the value of A to the prior state it recorded, £2000, nor should it do nothing. The problem is that the isolation property of transaction $T_1$ is violated. The result of $T_1$ has been seen at the low-level by $T_2$ which goes on to commit a value of A on this basis, thus we cannot simply undo $T_1$ by restoring the prior state recorded by $T_1$.

In this paper, we present a protocol called the dual-level validation method (DLV) which is used to provide atomicity for atomic data types; that is, atomicity of individual operations, serialisability of the transactions that use the objects and recoverability of the objects. The rest of the paper is organised as follows. In Section 2, we specify the DLV method informally. In Section 3, some preliminary definitions and lemmas are introduced. The DLV method is described formally and verified in Section 4. The recovery method is discussed in Section 5 and Section 6 describes our implementation of DLV. Section 7 concludes the paper and includes a comparison with related work.

2 The Dual-Level Validation Method

2.1 Atomic Objects

We view an atomic object as a two-layered architecture. The high layer, called the logical level, is a set of abstract operations defined on the object, which are the only means for users to access the object. The low layer, called the physical level, is a set of operations provided by the system to manage primitive data objects.

We use an optimistic approach to concurrency control. Transactions operate on shadow copies of (components of) objects, relying on commit-time validation to ensure serialisability. The two levels of DLV are concerned with the two levels of the object architecture,
physical and logical.

Physical level validation ensures that the logical level object operations are elementary. This level is concerned with four kinds of physical operation: create, delete, read and write. Logical level validation then ensures that the transaction that has used the object, and which is requesting commit, is serialisable with other transactions.

2.2 Transactions

A transaction, in general, encloses operations on several objects. The sequence of operations of a transaction on a particular object forms the component of the transaction at that object.

One approach to implementing optimistic concurrency control is to take a shadow copy of a whole object at the start of a transaction, or perhaps at the time of the first operation that updates the object. All subsequent invocations are on this copy and are validated against the persistent object when the transaction requests commit [Bac93].

Our objects are tree structured with primitive objects as leaves. We assume that objects may be large. Our approach is to take a shadow copy only of the subobject of the (tree structured) physical object that is required for a given invocation. A copy of a subobject is taken on the first invocation that updates it and all subsequent invocations on that subobject, for read or update, are performed on that shadow. A later invocation by the transaction may cause a shadow of a different subobject to be taken and this shadow may contain the committed updates of concurrent transactions.

An execution of a transaction consists of two, three or four phases: a read phase, a validation phase, and possibly a pending phase and a write phase (See Figure 1).

![Figure 1: The four phases of a transaction](image)

During the read phase, the transaction manager passes each operation enclosed in a transaction to the appropriate object. The object arranges immediate execution of the operation and records details of the invocation. If the invocation involves an update, this takes place at a local shadow copy of the physical subobject as described above. Each object therefore has a record of which object operations have been performed by each transaction, and which physical (sub)objects, together with their version numbers, have been read or written by each transaction.

The validation phase begins when the execution of a transaction reaches its end. During the validation, the transaction manager first assigns a timestamp to the transaction, and
then communicates with every object involved, passing it the transaction identifier and the timestamp. Each object validates its component of the transaction and indicates accepted or rejected. The aim is to establish whether any of the invocations of the transaction have been invalidated by the invocations of concurrent transactions, see Section 2.3. This stage is called logical validation. Note that we do not assume that logical validation takes place in timestamp order at each object.

Each accepted component of the transaction enters the pending phase with a "waiting" status, while each rejected component is aborted. Note that aborting simply involves discarding the shadow subobjects. The transaction manager then asks every object involved whether the component of the transaction handled at that object is accepted. If all are accepted, the transaction as a whole is committed, otherwise the transaction as a whole is aborted. The transaction manager informs every object involved of the result. If the result is commit, then the component at each object remains in the pending phase but with a new status "commit"; otherwise the component at each object is aborted and removed from the pending queue.

A component of a transaction in the pending phase does not necessarily enter the write phase immediately after the object gets the final result from the transaction manager. This is because there may be several pending components, associated with different transactions, at an object. They must enter the write phase in the order defined by their timestamps and, at any time, there is at most one component in the write phase at a particular object.

After entering the write phase, a component of a transaction is validated again by the object to check whether it can be accepted at the physical level. The purpose of this validation is to check whether the values read by the component are still up to date. If they are, the transaction is committed by merging its shadow copies into the permanent state. Otherwise the shadow copies are discarded and the operations of the component are re-executed. During the re-execution, any update to a physical object takes place in a shadow copy of the object as in the read phase. After the re-execution, shadow copies are merged into the permanent state.

2.3 Validation Algorithms

The purpose of logical validation in DLV is to ensure that the concurrent execution of a set of transactions is equivalent to executing these transactions serially in some order. To do this, each transaction $T_i$ is explicitly assigned a unique number $t_i$, called the timestamp of the transaction, at the end of the read phase. The validation algorithm then ensures that there exists a serially equivalent schedule in which transaction $T_i$ comes before transaction
$T_j$ whenever $t_i < t_j$. This can be guaranteed by the following validation condition [KR81, Pap79]. For each transaction $T_j$ with transaction number $t_j$, and for all $T_i$ with $t_i < t_j$, one of the following three conditions must hold (see Figure 2):

1. $T_i$ completes its write phase before $T_j$ starts its read phase.

2. The operation set of $T_i$ does not invalidate the operation set of $T_j$, and $T_i$ completes its write phase before $T_j$ starts its write phase.

3. Neither the operation set of $T_i$ invalidates the operation set of $T_j$ nor the operation set of $T_j$ invalidates the operation set of $T_i$, and $T_i$ completes its read phase before $T_j$ completes its read phase.

![Figure 2: Possible interleaving of two transactions](image)

The DLV method uses an algorithm that is an implementation of validation conditions 1 and 2. The first validation condition can be checked by recording the latest committed transaction’s timestamp when a transaction starts. Validating the second condition is done by the following three checks (suppose transaction $T_j$ is under validation):

- **Check 1**: For every transaction $T_i$ that is older than $T_j$, and had not committed when $T_j$ began, check whether the operation set of $T_i$ invalidates the operation set of $T_j$; if it does, the validation fails.

- **Check 2**: For every transaction $T_k$ that is in its pending phase and is younger than $T_j$, check whether the operation set of $T_j$ invalidates the operation set of $T_k$; if it does, the validation fails.

- **Check 3**: Check whether any committed transaction $T_k$ is younger than $T_j$; if any $T_k$ is, the validation fails.

**Check 1 and Check 2** ensure that the first part of validation condition 2 holds, *i.e.*, the operation set of an older transaction does not invalidate the operation set of a younger
transaction. Check 3 ensures that the second part of validation condition 2 holds, i.e.,
transactions are committed in the order defined by their timestamps. Note that transac-
tions need not enter the validation phase in their timestamp order. Check 2 and Check 3
ensure that the validation condition holds.

3 Preliminaries

3.1 Serial Dependency Relations

In this subsection, we briefly introduce the formal method developed by Weihl [Wei89]
and the serial dependency relation introduced by Herlihy [Her90].

Each object has a type which defines a state and a set of operations. An event is a
pair consisting of an operation invocation and a response. In the absence of failure and
concurrency, an object's state is modelled by a sequence of events called a history. A
specification for an object is the set of permissible histories for that object. A legal history
is one that is included in the object's specification.

In the presence of failure and concurrency, an object's state is given by a schedule,
which is a sequence of events, transaction commits, and transaction aborts. To keep track
of interleaving, a transaction identifier is associated with each step in a schedule. For
example, the following is a schedule for an account object:

$T_1$: credit(£800) / OK
$T_2$: credit(£1000) / OK
$T_1$: commit
$T_2$: debit(£1500) / OK
$T_2$: commit

(Serial) histories and (concurrent) schedules are related by the notion of atomicity. Let
$\leq$ denote a total order on committed and active transactions, and let $H$ be a schedule.
The serialisation of $H$ in the order $\leq$ is the history $h$ constructed by reodering the events
in $H$ so that if $T_1 \leq T_2$ then the subsequence of events associated with $T_1$ precedes the
subsequence of events associated with $T_2$. $H$ is serialisable in order $\leq$ if $h$ is legal. The
schedule in the example above is serialisable in order $T_1 \leq T_2$, but it is not serialisable in
order $T_2 \leq T_1$.

$H$ is serialisable if it is serialisable in some order. $H$ is atomic if the subschedule asso-
ciated with committed transactions is serialisable. An object is atomic if it only produces
atomic schedules.

We wish to take account of all events that might, directly or indirectly, have influenced
e. Let $<_d$ be a relation between pairs of events, and let $h$ be a history. A subhistory (i.e.,
a subsequence) $g$ of $h$ is closed under $<_d$ if whenever it contains an event $e$ it also contains
every event $e'$ of $h$ such that $e' <_d e$.

A subhistory $g$ is a view of $h$ for $e$ under $<_d$ if $g$ is closed under $<_d$, and if $g$ contains every $e'$ of $h$ such that $e' <_d e$.

Informally, $<_d$ is a serial dependency relation if whenever an event is legal for a view, it is legal for the complete history. More precisely, let "•" denote concatenation:

**Definition 1** A relation $<_d$ is a serial dependency relation if $g \bullet e$ is legal implies that $h \bullet e$ is legal, for all events $e$ and all legal histories $g$ and $h$, such that $g$ is a view of $h$ for $e$ under $<_d$.

We make use of the following lemma proved in [Her90] when reasoning about serial dependency relations. It states that any sequence of events can be inserted into the middle of a history provided no later event depends on any inserted events.

**Lemma 1** If $<_d$ is a serial dependency relation, $f$, $g$ and $h$ histories such that $f \bullet g$ and $f \bullet h$ are legal, and there is no $e$ in $g$ and $e'$ in $h$ such that $e <_d e'$, then $f \bullet g \bullet h$ is legal.

Proof: The proof is by induction on the length of $h$. If $h$ is empty, the result is immediate. Otherwise, let $h = h' \bullet e'$. By assumption, $f \bullet h'$ is a view of $f \bullet g \bullet h'$ for $e'$. Moreover, $f \bullet g \bullet h'$ is legal by the inductive hypothesis and $f \bullet h'$ is legal because $f \bullet h$ is legal by assumption. Because $f \bullet g \bullet h'$ is legal and $<_d$ is a serial dependency relation, $f \bullet g \bullet h' \bullet e' = f \bullet g \bullet h$ is legal by Definition 1. □

### 3.2 Views of a Transaction

We formalise an operation on an object as a function that reads from some primitive physical objects (maybe none), and based on the results of reading and its parameters writes to some primitive physical objects (maybe none). We use $o(R, W)$ to denote an operation which reads from a set of primitive physical objects $R = [r_0, \ldots, r_m]$ and writes to a set of primitive physical objects $W = [w_0, \ldots, w_n]$.

Internally, an object is implemented by two components: a permanent state that records the effect of committed transactions, and a set of local versions (shadow copies) that record each active transaction's tentative changes. Let $T = [T_0, \ldots, T_n]$ be a set of transactions, $d_u$ be a primitive physical object, we use $d_u^i$ to denote a specific version $i$ (created for transaction $T_i$) of $d_u$, and $d_u^p$ to denote its permanent state. To process operations from $T_i$, an object must translate an operation of $T_i$ on a (single version) primitive physical object into an operation on a specific version of that physical object. This translation is formalised by a function $tr$.

**Definition 2** Let $o(R, W)$ be an operation from transaction $T_i$, $R = [r_0, \ldots, r_m]$, $W = [w_0, \ldots, w_n]$, $R' = [r_0^0, \ldots, r_m^0]$, $W' = [w_0^0, \ldots, w_n^0]$, then $tr(o(R, W)) = o(R', W')$, where
1. $k_u = i$ for $0 \leq u \leq n$;

2. $j_u = i$ for $0 \leq u \leq m$, if an operation of $T_i$ has written to $w_u$;

3. $j_u = p$ for $0 \leq u \leq m$, if no operation of $T_i$ has written to $w_u$.

Rule 1 states that a transaction can only write to its own version of physical objects. Rule 2 states that if a version of a physical object has been created for a transaction, then it must read from that version. Rule 3 states that if a transaction has not written to a physical object, it must read from the permanent state of the object.

The permanent state of an object can be modelled by a sequence of histories each of which is a sequence of events caused by committed transactions. More precisely,

**Definition 3** Let $C = o_1 \cdots o_n$ be the component of transaction $T_i$ at an object and $h_j$ denote the change in the permanent state of the object between the execution of $o_j$ and $o_{j+1}$, then the permanent state of the object when executing $o_l (1 \leq l < n)$ is $p_{s_l}(T_i) = h_0 \cdots h_{l-1}$, where $h_0$ is the object state before executing $o_1$.

An object state consists of a set of primitive physical objects, the leaves in the tree structure. A change on an object $D$ made by an operation can be represented by corresponding changes on $D$'s primitive physical objects. A read from $D$ issued by an operation can be represented by corresponding reads from $D$'s primitive physical objects. If we denote an object $D$ consisting of a set of primitive physical objects $d_1 \cdots d_m$ as:

$$D = \begin{pmatrix} d_1 \\ \vdots \\ d_m \end{pmatrix},$$

then an operation $o$ on $D$ can be represented by a group of operations on the primitive physical objects:

$$o = \begin{pmatrix} o^1 \\ \vdots \\ o^m \end{pmatrix},$$

where $o^i$ is an operation on physical object $d_i$. We call $o^i$ a suboperation of $o$ on physical object $d_i$. An event $e$ on $D$ can be represented by a group of events:

$$e = \begin{pmatrix} e^1 \\ \vdots \\ e^m \end{pmatrix},$$

where $e^i$ is a subevent of $e$ on $d_i$. Furthermore, a history $h$ for $D$ can be denoted as:

$$h = e_1 \cdots e_n = \begin{pmatrix} e^1_1 \cdots e^1_n \\ \vdots \\ e^m_1 \cdots e^m_n \end{pmatrix} = \begin{pmatrix} h^1 \\ \vdots \\ h^m \end{pmatrix}.$$
We call \( e_1^i \cdots e_n^i \) the subhistory of \( h \) on physical object \( d_i \), denoted by \( h^i \). The history for a physical object \( d_j \), \( h^j = e_1^j \cdots e_n^j \), is locally legal if it is the same as executing the corresponding operations on \( d_j \) in a sequential environment in the same order.

**Lemma 2** Suppose object \( D \) consists of a set of primitive physical objects \( d_1 \cdots d_m \), \( h = e_1 \cdots e_n \) is the history for \( D \). Then \( h \) is legal if and only if every subhistory of \( h \) for each primitive physical object is locally legal.

**Proof:** At first we prove that if \( h \) is legal, then every subhistory of \( h \) for a physical object is locally legal. Since \( h = e_1 \cdots e_n \) is legal, \( h \) must be a permissible history for the object. That is, \( h \) must be the same as executing the corresponding operations on \( D \) in a sequential environment. Hence, every subhistory \( h^j \) of \( h \) must be the same as executing the corresponding suboperations on \( d_j \) in a sequential environment. Therefore, \( h^j \) is locally legal by its definition.

Now let's prove that if every subhistory of \( h \) for a physical object is locally legal, then \( h \) is legal. Since every subhistory, \( h^j = e_1^j \cdots e_n^j \), is locally legal, it is the same as executing the corresponding suboperations on \( d_j \) in a sequential environment in the same order. Moreover, \( e_i \) is the composite of \( e_1^i \cdots e_{m_i}^i \). Therefore, \( h \) must be the same as executing the corresponding operations on \( D \) in a sequential environment. Hence, \( h \) must be legal. \( \square \)

A view of a transaction \( T_i \) for an (abstract) object, denoted by \( \text{View}(T_i) \), is the value of the object that \( T_i \) observes at some moment. More precisely,

**Definition 4** Let \( C = a_1 \cdots a_n \) be the component of transaction \( T_i \) at an object and \( e_j \) be an event of executing operation \( a_j \) \( (0 \leq j \leq n) \), suppose that the object state consists of primitive physical objects \( d_1 \cdots d_m \) and we have \( ps_j(T_i) = h_0 \cdots h_{j-1} \). Then after executing operation \( a_j(1 \leq j \leq n) \), we have

\[
\text{View}(T_i) = \begin{pmatrix}
 v^1 \\
 \vdots \\
 v^m
\end{pmatrix} = \begin{pmatrix}
 h_0^1 \cdots h_{k_1-1}^1 \cdots e_1^j \\
 \vdots \\
 h_0^m \cdots h_{k_m-1}^m \cdots e_m^j
\end{pmatrix}
\]

where \( e_u^i \) is the subevent of \( e_u \) on physical object \( d_i \) \( (1 \leq l \leq m, k_i \leq u < j) \). Here, \( e_{k_l}^i \) \( (1 \leq l \leq m) \) is the first subevent that writes physical object \( d_i \). We call \( v^i \) the subview of \( \text{View}(T_i) \) on physical object \( d_i \).

**Lemma 3** Suppose \( \text{View}(T_i) \) is a view of transaction \( T_i \) for an object. Then every subview of it is locally legal, if the permanent state of the object is legal.
Proof: Let's consider the subview for $d_i$, $v^i = h_0^i \cdots h_{k_i-1}^i \cdot e_{k_i}^i \cdots e_j^i$. Since the permanent state of an object is legal by assumption, $h_0^i \cdots h_{k_i-1}^i$ is locally legal. By Definition 2, we know that when executing operation $o_{k_i}$, the object creates a local version of $d_i$, which has the value $h_0^i \cdots h_{k_i-1}^i$, and all the subsequent operations of $T_i$ on $d_i$ are done on this local version. Since the local version can only be accessed by $T_i$, the event sequence $e_{k_i}^i \cdots e_j^i$ happened in a sequential environment. Therefore, $v^i = h_0^i \cdots h_{k_i-1}^i \cdot e_{k_i}^i \cdots e_j^i$ is locally legal. \qed

Now we can get the lemma that is important to the proof of the DLV method.

Lemma 4 Let $<_d$ be a serial dependency relation, $C = o_1 \cdots o_n$ be the component of transaction $T_i$ at an object, $e_i$ be the execution of $o_i$, $h_0$ be the object state before $e_1$, $h_j$ $(1 \leq j < n)$ be the change that happened at the object between $e_j$ and $e_{j+1}$, $h_n$ be the change that happened after $e_n$. Then $h_0 \cdots h_n \cdot e_1 \cdots e_n$ is legal, if $h_0 \cdots h_n$ is legal and if there is no $e$ in $h_1 \cdots h_n$ and $e'$ in $e_1 \cdots e_n$ such that $e <_d e'$.

Proof: The proof is by induction on the length of $C$, that is, the number of its operations. If the length $n = 1$, by Definition 2, $h_0 \cdot e_1$ is legal. By assumption, $h_0 \cdot h_1$ is legal and there is no event $e$ in $h_1$ such that $e <_d e_1$. That is, $h_0$ is a view of $h_0 \cdot h_1$ for $e_1$. Therefore, $h_0 \cdot h_1 \cdot e_1$ is legal by Definition 1.

If the length $n > 1$, then

1. (a) by assumption, $h_0 \cdots h_{n-1} \cdot h_n$ is legal,
   
   (b) by the inductive hypothesis, $h_0 \cdots h_{n-1} \cdot e_1 \cdots e_{n-1}$ is legal, and
   
   (c) by assumption, there is no event $e$ in $h = h_n$ and $e'$ in $h = e_1 \cdots e_{n-1}$ such that $e <_d e'$.

Therefore, $p = h_0 \cdots h_{n-1} \cdot h_n \cdot e_1 \cdots e_{n-1}$ is legal by Lemma 1. Consequently, every $p^j = h_0^j \cdots h_n^j \cdot e_1^j \cdots e_{n-1}^j$ is locally legal by Lemma 2.

2. (a) By Definition 4, there is no event $e_j$ in $e_1^j \cdots e_{k_j-1}^j$ that writes $d_j$. Therefore, there is no event $e_j$ in $e_1^j \cdots e_{k_j-1}^j$ such that $e_j <_d e_j^i$. This is because in the physical level there is only one serial dependency relation: write $<_d$ read.

(b) By assumption, there is no $e$ in $h_{k_j} \cdots h_n$ such that $e <_d e_n$. Notice that if nonsense can arise at a physical level, the user must declare the potential nonsense in the abstract semantics. Hence we know that there is no $e_j$ in $h_{k_j} \cdots h_n$ such that $e_j <_d e_j^i$. 

10
Therefore, \( q^i = h_0^i \bullet \cdots \bullet h_{k_j - 1}^i \bullet e_{k_j}^i \bullet \cdots \bullet e_{n-1}^i \) is a view of \( p^i \) for \( e_n^i \).

3. Since \( h_0 \bullet \cdots \bullet h_n \) is legal by assumption, every subview of \( \text{View}(T_i) \) is locally legal by Lemma 3. Hence \( v^i = q^i \bullet e_n^i = h_0^i \bullet \cdots \bullet h_{k_j - 1}^i \bullet e_{k_j}^i \bullet \cdots \bullet e_{n-1}^i \) is locally legal.

Therefore, \( p^i \bullet e_n^i = h_0^i \bullet \cdots \bullet h_{k_j - 1}^i \bullet h_{k_j}^i \bullet e_1^i \bullet \cdots \bullet e_{n-1}^i \bullet e_n^i \) for \( 1 \leq j \leq m \) is locally legal by Definition 1. Hence \( h_0 \bullet \cdots \bullet h_n \bullet e_1 \bullet \cdots \bullet e_n \) is legal by Lemma 2. \( \square \)

4 The Correctness of the DLV Method

4.1 The Dual-Level Validation Automaton

Formally, each object is modelled by an automaton that accepts certain schedules. The automaton's state is defined using the following primitive domains: TRANS is the set of transaction identifiers, DIDS is the set of physical object identifiers, EVENTS is the set of events, and TIMESTAMP is a totally ordered set of timestamps. The derived domain HISTORY is the set of sequences of events. A dual-level validation automaton has the following state components:

<table>
<thead>
<tr>
<th>Perm:</th>
<th>HISTORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>View:</td>
<td>TRANS ( \rightarrow ) HISTORY</td>
</tr>
<tr>
<td>Intentions:</td>
<td>TRANS ( \rightarrow ) HISTORY</td>
</tr>
<tr>
<td>ReadSet:</td>
<td>TRANS ( \rightarrow ) DIDS</td>
</tr>
<tr>
<td>ReadVersion:</td>
<td>(TRANS, DIDS) ( \rightarrow ) TIMESTAMP</td>
</tr>
<tr>
<td>WriteSet:</td>
<td>TRANS ( \rightarrow ) DIDS</td>
</tr>
<tr>
<td>WriteVersion:</td>
<td>(TRANS, DIDS) ( \rightarrow ) TIMESTAMP</td>
</tr>
<tr>
<td>TimeStamp:</td>
<td>TRANS ( \rightarrow ) TIMESTAMP</td>
</tr>
<tr>
<td>BeginTime:</td>
<td>TRANS ( \rightarrow ) TIMESTAMP</td>
</tr>
<tr>
<td>CommitTime:</td>
<td>TRANS ( \rightarrow ) TIMESTAMP</td>
</tr>
<tr>
<td>Version:</td>
<td>DIDS ( \rightarrow ) TIMESTAMP</td>
</tr>
<tr>
<td>LastCommitTime:</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>Clock:</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>Committed:</td>
<td>(2^{\text{TRANS}})</td>
</tr>
<tr>
<td>Aborted:</td>
<td>(2^{\text{TRANS}})</td>
</tr>
</tbody>
</table>

The DLV automaton enforces the atomicity of schedules generated at the object. Its behaviour is specified by giving the transitions during the read phase and the write phase.

Each transition has a precondition and a postcondition. In postconditions, primed component names denote new values, and unprimed names denote old values.
At the read phase:

For a transaction $T_i$ to execute operation $o(R,W)$ at an object, where $R = [r_1, \ldots, r_m]$, $W = [w_1, \ldots, w_n]$:

Pre: $T_i \notin Committed \cup Aborted$.

$e$ is the event of executing $o(R,W)$.

Post: $View'(T_i) = View(T_i) \cup o(R,W)$

$Intentions'(T_i) = Intentions(T_i) \cdot e$

$ReadSet'(T_i) = ReadSet(T_i) \cup [r_1, \ldots, r_m]$

$WriteSet'(T_i) = WriteSet(T_i) \cup [w_1, \ldots, w_n]$

$BeginTime'(T_i) = \min(BeginTime(T_i), \text{Clock})$

$ReadVersion'(T_i, r_j) = \begin{cases} 
  \text{ReadVersion}(T_i, r_j) & \text{if } r_j \in \text{ReadSet}(T_i) \\
  \text{Version}(r_j) & \text{otherwise}
\end{cases}$

$WriteVersion'(T_i, w_j) = \begin{cases} 
  \text{WriteVersion}(T_i, w_j) & \text{if } w_j \in \text{WriteSet}(T_i) \\
  \text{Version}(w_j) & \text{otherwise}
\end{cases}$

where "$w$" denotes executing an operation according to Definition 2.

The DLV automaton does not undergo any transition during the validation phase. The result of logical validation is reported to the DTM, which returns a decision commit or abort for the transaction. Committed transactions will subsequently enter the write phase. Logical validation at an object is governed by a conflict relation $<_c$ defined at the object.

**Definition 5** A transaction $T_i$ is logically valid for relation $<_c$ on an object, if the following three conditions hold:

- For each transaction $T_j$ such that $\text{TimeStamp}(T_j) < \text{TimeStamp}(T_i)$ and $\text{BeginTime}(T_i) < \text{CommitTime}(T_j)$ (committed earlier transactions), there is no $e$ in $\text{Intentions}(T_i)$ and no $e'$ in $\text{Intentions}(T_j)$ such that $e' < _c e$.

- For each transaction $T_k$ such that $\text{TimeStamp}(T_i) < \text{TimeStamp}(T_k)$ and $T_k$ is in its pending phase (pending later transactions), there is no $e$ in $\text{Intentions}(T_i)$ and no $e'$ in $\text{Intentions}(T_k)$ such that $e < _c e'$.

- $\text{TimeStamp}(T_i) > \text{LastCommitTime}$ (commit in timestamp order).

When a committed transaction proceeds to the write phase we perform physical validation to check whether we can avoid re-executing the operations. If so, then the transaction
is committed by using its view for the object to replace the permanent state of the object; otherwise we must apply Intentions($T_i$) to the permanent state of the object.

Physical validation is done by checking whether the version number of each physical object in the read set of a transaction is still current. More precisely,

**Definition 6** A transaction $T_i$ is physically valid at an object if there is no physical object $d$ in ReadSet($T_i$) such that Version($d$) > ReadVersion($T_i$, $d$). That is, the value of $d$ read by $T_i$ is still current.

At the write phase:

Depending on the result of physical validation transaction commitment is defined by the following transition of the DLV automaton.

If physical validation succeeds:

Pre:

- $T_i \not\in$ Committed $\cup$ Aborted.
- $T_i$ is logically valid.
- $T_i$ is physically valid.
- TimeStamp($T_i$) > LastCommitTime.

Post:

- Perm' = View($T_i$)
- Clock' > Clock
- Version'($d^i$) = Clock, for any $d^i \in$ WriteSet($T_i$);
- LastCommitTime' = TimeStamp($T_i$)

If physical validation fails:

Pre:

- $T_i \not\in$ Committed $\cup$ Aborted.
- $T_i$ is logically valid for relation $<_e$.
- TimeStamp($T_i$) > LastCommitTime;

Post:

- Perm' = Perm $\bullet$ Intentions($T_i$)
- Clock' > Clock
- Version'($d^i$) = Clock, for any $d^i \in$ WriteSet($T_i$);
- LastCommitTime' = TimeStamp($T_i$)

13
4.2 Atomicity of the DLV Method

To verify the dual-level validation method, a new concept needs to be introduced. We want to define equivalence so that two histories of an object are equivalent if they have the same effects on the object. The effects of a history on an object are the values produced by write operations in the history.

Definition 7 Two histories of an object are view equivalent if they produce the same object value, denoted by “≡”.

Two histories of a physical object are equivalent if the last writes to the object in the two histories are of the same value. An operation is a function that reads some values from and writes some values produced by it to an object. If an operation reads the same values in two histories, then the values it writes will be the same in the two histories.

Lemma 5 If View(Ti) is the view of Ti for an object when it entered the write phase, Perm is the permanent state of the object, then View(Ti) ≡ Perm • Intentions(Ti) for any physically valid transaction Ti.

Proof: Suppose the view of Ti for an object when it entered the write phase is

\[ \text{View}(T_i) = \left( \begin{array}{c} h^1_0 \cdots h^1_{k_1-1} \cdots c^1_{k_1} \cdots c^1_n \\ \vdots \\ h^m_0 \cdots h^m_{k_m-1} \cdots c^m_{k_m} \cdots c^m_n \end{array} \right) \]

Since Ti is physically valid, for any d_j either it does not belong to the read set of Ti or it has not been changed after Ti read it. That is, there is no \( e^j \) in \( h^i_{k_j} \cdots h^i_n \) and \( e^j \) in \( e^j \cdots e^j_n \) such that \( e^j \) writes \( d_j \) and \( e^j \) reads \( d_j \). Moreover, by Definition 2, \( e^1_i, \ldots, e^j_{k_j-1} \) are each either a read only event or an empty event, otherwise a shadow copy would have been created. Thus, \( h^j \equiv h^i_0 \cdots h^i_{k_j-1} \cdots h^j_n \cdots e^j_i \cdots \cdots e^j_{k_j-1} \cdots e^j_{k_j} \cdots e^j_n \), because \( e^j \cdots e^j_n \) reads the same values, thus writes the same values in both histories. Therefore, \( \text{View}(T_i) \equiv h_0 \cdots h_n \cdot e_1 \cdots e_n = \text{Perm} \cdot \text{Intentions}(T_i) \). □

Notice that in the DLV method, at any time there is at most one transaction in the write phase at a particular object. Therefore, the view of a transaction for an object will be the same throughout the write phase.

Lemma 6 For any dual-level validation automaton whose logical validation relation \( <_c \) is a serial dependency relation, \( \text{Perm} \cdot \text{Intentions}(T_i) \) is legal for any logically valid Ti.

Proof: Suppose \( C = o_1 \cdots o_n \) is the component of Ti at the object. We may now write Perm = \( h_0 \cdots \text{Intentions}(T_i) \cdots \text{Intentions}(T_{i-1}) = h_0 \cdots h_n \), and Intentions(Ti) = \( e_1 \cdots e_n \), where \( T_1, \ldots, T_{i-1} \) have been committed with timestamp earlier than that of
$T_i$. The proof is by induction on the number of transactions that have entered the write phase before $T_i$.

When $i = 1$, $\text{Perm} \cdot \text{Intentions}(T_1) = h_0 \cdot \text{Intentions}(T_1)$: it is legal by Lemma 4 because $h_1 \cdot \cdots \cdot h_n$ is empty.

When $i > 1$, by the inductive hypothesis, $h_0 \cdot \text{Intentions}(T_1) \cdot \cdots \cdot \text{Intentions}(T_{i-1})$ is legal. Since $T_i$ is logically valid by assumption, there is no $e$ in $\text{Intentions}(T_1) \cdot \cdots \cdot \text{Intentions}(T_{i-1})$ and $e'$ in $e_1 \cdot \cdots \cdot e_n$ such that $e <_c e'$. Moreover, $<_c$ is a serial dependency relation. Therefore, $h_0 \cdot \text{Intentions}(T_1) \cdot \cdots \cdot \text{Intentions}(T_{i-1}) \cdot \text{Intentions}(T_i)$ is legal by Lemma 4. □

**Lemma 7** The dual-level validation method is atomic, if the conflict relation used by the logical validator is a serial dependency relation.

**Proof:** From the dual-level validation automaton, we know that the permanent state of an object is the serialisation in timestamp order of the schedule accepted by the automaton. Moreover, Lemma 5 and Lemma 6 imply that each commit carries the permanent state from one legal history to another. Therefore, the schedule is atomic. Since its automaton only accepts atomic schedules, the DLV method is atomic. □

5 Recovery

To ensure data consistency, a system needs to provide three kinds of recovery [CP84]. The activity of ensuring a transaction's atomicity in the presence of transaction aborts is called transaction recovery. The activity of ensuring a transaction's atomicity in the presence of system crashes, in which only volatile storage is lost, is called crash recovery. The activity of providing a transaction's durability in the presence of media failures, in which nonvolatile storage is lost, is called database recovery.

In the introduction we pointed out that a transaction's isolation property may be violated in an implementation of transactions based on atomic data types. This results in the failure of traditional state-based recovery. DLV uses optimistic concurrency control in which a transaction performs updates operations on local copies of objects during its read phase. Transaction abort is therefore achieved by discarding these shadow copies. Persistent object values are not affected until the write phase of a transaction.

Database recovery is independent of the concurrency control method used by a transaction system. Various methods such as stable storage [Lam81] can be used for providing database recovery.

Our crash recovery method is log-based [Gra79], but in a system which uses DLV to provide local atomicity there is no need to write a log record for an object operation. This
is because, in this method, updates to an object can only be made on its shadow copies before a transaction commits, see Section 2.3.

A log record is recorded on stable storage when each phase of a transaction starts, and when a transaction completes (aborts or commits). Hence during a recovery procedure after a system crash, the status of a transaction can be determined.

If a transaction was in its read phase when the system crashed, it will be aborted when the system restarts. No special recovery operation needs to be done, since the transaction neither made any change to a persistent object, nor made any promise. Any local copy of objects it has created will be collected by the garbage collector.

Before entering the validation phase, the performed-operations table (POT) and the accessed-objects table (AOT) of the transaction must be recorded on stable storage. If a transaction was in its validation phase when the system crashed, the object will do the validation again at restart. The information necessary for the validation, i.e. the POT, has been recorded on stable storage.

If a transaction was in its pending phase when the system crashed, it will remain in this phase at restart. No special action needs to be taken. However the pending queue of an object which records all the transactions in their pending phase needs to be refreshed to stable storage whenever a change is made to it.

The write phase of a transaction is separated into two or three steps: a physical validation step, possibly a re-execution step, and a merging step. A log record is necessary to indicate the end of a step. Moreover, if a re-execution step is required, the new POT and AOT produced by the re-execution need to be recorded on stable storage before the merging step.

If a transaction was in its physical validation step when the system crashed, at restart the object will perform physical validation again for that transaction. The validation can be performed because the AOT with the required information has been written to stable storage. If a transaction was in the merging step, at restart the object will redo the merging operation. This can be done because all the shadow copies as well as the AOT have been written to stable storage. Notice that a merging operation is idempotent. If a transaction was in the re-execution step, at restart the object will re-execute the operations on the object of the transaction. The re-execution can be done since the POT which recorded the operations of the transaction has been written to stable storage.

6 An Implementation

The DLV method has been implemented and works well in a persistent programming language PC++ [Wu93, WMB93] which is a persistent extension of C++. In this section
we show how the DLV method is used to implement atomic data types in PC++. 

In order to construct an atomic object a programmer must not only specify the object representation and object operations but also must implement the functionality of local atomicity. This is a difficult task. In order to lessen the programmer's burden, PC++ takes an implicit approach to implementing atomic data types. A special type called *Scheduler* is available which implements the DLV method. To provide local atomicity, user-defined atomic types inherit this method from the *Scheduler* by making use of type inheritance. The semantics of object operations are specified by users in the form of conflict relations. Logical validation of an object is done according to the conflict relation of the object.

The state of an atomic object is represented by a number of physical objects. The durability of transactions requires that object states modified by transactions become permanent when they commit. To achieve this, PC++ uses the services of a multi-service storage architecture (MSSA) for the storage of physical objects.

### 6.1 The MSSA

The Opera group in the Computer Laboratory at Cambridge has designed a storage architecture, the MSSA, to support multi-media applications [BMTW91, MBB+93]. As well as traditional and continuous media files, among the file types recognised are 'structured files' which can include references to any MSSA object. PC++ uses the 'Structured File Custode' (SFC) [Tho90] to provide storage for the physical objects.

The SFC provides a large, shared, persistent object store, directly accessible from programming languages. An important feature of the SFC is that it supports the storage of *structured object representations*; that is, a highly structured object can be represented directly by the SFC. The SFC is not a type manager, being concerned only with the primitive storage types *byte* and *storage service identifier*. User programs can access objects at any abstraction granularity, from a basic field such as an integer or char (these types are known only to the programming language) to a whole object. Therefore, when making changes to a component of an object only that component needs to be rewritten; no other component of the object is affected. The SFC provides a multiple granularity locking mechanism as described by Gray [Gra79], which can be used to lock any logical component of an object.

The use of the SFC for data storage has proved very convenient. Since SFC objects are tree-structured data migration and shadow versions may be managed at subobject level, which meets the requirement of the DLV method. The multiple granularity locking mechanism provided by the SFC means that concurrency control can be applied at any granularity required by the DLV method.
6.2 The Scheduler

6.2.1 Operations

The Scheduler is an implementation of the DLV method. It provides five public operations: create, invoke, validate, object.abort and object.commit. These operations are used by the transaction manager to communicate with an atomic object. By using inheritance, these operations can become the properties of a user-defined atomic data object.

Before any operation can be executed on it, an atomic object must be activated. This can be done in either of the following ways: by calling the invoke operation, if the object exists; or by calling the create operation, otherwise. After an atomic object is activated, the calling transaction is registered with the object, so that it can call operations defined on the object.

The operation validate is called when the transaction manager intends to ask participating objects to vote on a transaction. The validate operation performs logical validation according to the conflict relation of the object. When the transaction manager has decided to commit a transaction, it should ask every participating object to commit that transaction locally by invoking the object.commit operation. This operation does the work of the write phase manager (WPM). The object.abort operation is responsible for aborting a transaction locally. This can be done simply by discarding the shadow copies created for the transaction.

6.2.2 Data Members

Scheduler also defines a number of state variables to record information about transactions that share an object. A state variable event_table is used by an object to record the operations that are performed on the object by each transaction together with their parameters and results. The state variables read_set, write_set, create_set and delete_set are used to record the physical objects which are read, written, created, or deleted by a transaction respectively. Further, a state variable lastcommitted is used to record the timestamp of the latest committed transaction.

State variables pending_queue and committed_queue play a very important role in the DLV method. They are used by both the logical validator (LOV) and the write phase manager (WPM). An important property of the DLV method is that transactions are committed in timestamp order. This property could be achieved by enforcing that at every object transactions are validated in their timestamp order and by implementing the validation phase and the write phase together as a single atomic operation. However, this would reduce greatly object concurrency and availability. Therefore, it is desirable,
especially in a distributed transaction system, to permit transactions to be validated in an arbitrary order and to separate the validation phase from the write phase. We realise this by using the `pending_queue` and `committed_queue`.

The algorithm works in the following way. The LOV validates a transaction: if validation succeeds, it puts the transaction in the `pending_queue` with status `valid` and begins to validate another transaction; if validation fails, the transaction is aborted. When receiving the final decision about a transaction, the cooperation manager (COM) sets the transaction’s status to `commit` or `abort` accordingly. Meanwhile, the WPM checks the `pending_queue` from time to time to see whether the status of the transaction at the head of the `pending_queue` has become `abort` or `commit`. If it has, the WPM aborts or commits it, then removes it from the `pending_queue`. If a transaction is committed, it is put in the `committed_queue`.

This implementation ensures that transactions are committed in timestamp order, although it permits transactions to be validated in an arbitrary order. This is because the WPM only commits a transaction when it reaches the head of the `pending_queue`, and transactions are maintained in the queue in their timestamp order.

Furthermore, this implementation makes the following three actions independent: validating a transaction; getting the final decision about a transaction; and applying the updates of a transaction. As soon as the LOV has finished validating one transaction, it can begin validating another without needing to wait for the completion of the first. A transaction that has got its final decision but has not become the head of the pending queue needs to be held until all transactions in front of it have been completed. It is worth pointing out, however, that the application program does not need to wait for the completion of a transaction. It can continue its work immediately after receiving the final decision about the transaction.

### 6.3 Validating a Transaction

To validate a transaction $T$, the LOV needs to check whether other transactions have invalidated $T$. Two kinds of transactions may invalidate $T$: transactions that committed after $T$ began, and transactions that were validated after $T$ began but have not yet committed and are older than $T$. The first kind of transaction should have been recorded in the `committed_queue`, and the second kind of transaction in the `pending_queue`.

The LOV also needs to check whether $T$ would invalidate any transaction that has already passed its validation. Transactions that may be invalidated by $T$ are those that have passed their validations but have not yet committed and are younger than $T$.

Finally, the LOV needs to check whether the latest committed transaction is younger
than T. If it is, the validation fails because transactions must be committed in their timestamp order. This check can be done by comparing T's timestamp with the last_committed variable.

To check whether a transaction T1 may invalidate another transaction T2, the LOV simply needs to check whether any event in the event_table of T1 may invalidate any event in the event_table of T2 according to the conflict relation defined for the object.

6.4 Recording Events

One responsibility of the Read Phase Manager (RPM) is to record the events of a transaction into its event_table. The information in this table is necessary for validating and re-executing the transaction.

Since, in our implementation, transactions invoke object operations directly and the results of operations are returned to transactions directly, recording the events of a transaction must be done by the operations themselves. However, providing concurrency transparency is an aim of our design so it is inappropriate to ask programmers to write the code to perform the recording work for every object operation. A preprocessing method is therefore adopted to solve this problem.

During preprocessing, the preprocessor adds to every object operation some code which records the operation's name, parameters and results into the event_table whenever the operation is executed. It is easy for the preprocessor to find out the name and parameters of an operation by analysing its header. However, it is impossible for the preprocessor to get the results of an operation without the help of programmers. Fortunately, results of operations need only to be distinguished as succeeded or failed. Therefore, if programmers can tell the preprocessor whether a return point of an operation is a successful one or unsuccessful one, the preprocessor can add appropriate code at the return point to record the result. Programmers can do the job simply by writing an unsuccessful return in the form of "fail_return" instead of "return".

Under this implementation, therefore, whenever an operation is invoked by a transaction, it will automatically record its name, its parameters and its results into the event_table associated with that transaction.

7 Related Work and Conclusions

Several papers [Ton89, SS84, BGL83] have addressed the problem of extending concurrency control protocols to cope with arbitrary user-defined operations. These focus exclusively on locking protocols and do not consider recovery issues. In [Wei84, LCJS87, SDP91, SBD+85, AM83] implementations of atomic data types are described, but all of them
use pessimistic concurrency control methods and support elementary operations by using exclusive locks. Such methods limit concurrency at the low-level by forcing operations on an object to run serially. The DLV method separates the concerns and permits maximum concurrency at the low level while allowing the high level to focus exclusively on operation semantics. Also, our atomic objects are recoverable. A transaction can affect the object state only when it commits and the invocations on shadow copies can simply be discarded.

Like the DLV method, multi-level transactions[HW91, Wei91] increase concurrency by exploiting the semantics of high-level operations. The major difference here is that DLV is a single-level transaction. It is therefore cheaper to implement because only one level of recovery is required. DLV is most similar to the method proposed by Herlihy[Her90] in that both of them are optimistic and both of them use the semantics of operations to validate interleaving of invocations by transactions. However, there are significant differences between the methods.

Herlihy's method represents the partial results of a transaction by a snapshot of the permanent state of the object plus an intentions-list, and commits a transaction by applying the intentions-list serially to the permanent state. The DLV method represents the partial results of a transaction by a group of shadow copies of physical objects, and commits a transaction component by merging the shadow copies into the permanent state. When taking a snapshot of an object for a transaction, Herlihy's method is to create a copy of the whole object state, even if the transaction only accesses a small part of it. The shadow copies used by DLV are only of the components of objects that are required for the requested invocations. Our storage service architecture contains a structured data server which supports this well. Applications can therefore carry out transactions which involve very large objects without consuming system resources unnecessarily. DLV does not require that transactions go through logical validation serially in timestamp order. This feature is important for a distributed transaction system in which it is possible that an older transaction may request logical validation, at some of the objects involved, later than a younger one. Without this feature either an atomic, distributed validation algorithm would be needed to enforce timestamp ordering at all objects or out-of-order validation requests could be rejected by the objects. Both of these approaches are inferior to that adopted in DLV. Also, in DLV an object may begin validating one transaction as soon as it has finished another, without waiting for its completion. These two features make DLV suitable for distributed environments. It is particularly suited to a multimedia environment (the original motivation for the method) in which conflict is rare and real time requirements must be met.

At commit Herlihy's method re-executes the operations of the transaction at the per-

21
sistent object. Since the re-executions must be done serially, transactions may be delayed, waiting to commit. DLV requires physical validation, since transactions may conflict at the physical level even if they do not at the logical level. Transactions that pass physical validation are committed simply by merging the shadow copies into the permanent state. Transactions that fail physical validation are re-executed locally, as in Herlihy’s method, before being committed. Absence of conflict has already been established at the logical level.

The feasibility of the DLV method has been shown by our PC++ implementation. No special difficulty has arisen during its implementation. PC++ has already been used to reengineer a simple distributed application, namely to maintain the database for an active badge system that is used within the Laboratory. This is a low bandwidth application in which updates to the database are obtained from distributed collection points. The database can be interrogated from any terminal within the Laboratory.

PC++, including DLV, was developed within the Opera project [MBB+93]. It gives a programming language interface to the storage services which support the requirements of multimedia as well as conventional storage. Its optimistic approach, with no delay on access to objects, makes it ideal for programming reliable, distributed, multimedia applications and it will be evaluated in this context.

Acknowledgements

We acknowledge SERC support for this work under grant GR/H 13666 and ICL support of Z Wu. Thanks to members of the OPERA project and to Heather Brown, visiting this year, for many discussions on all aspects of the work.

References


