Proving Correctness of a Distributed Shared Memory Implementation

Miguel Castro*

castro@lcs.mit.edu

January 4, 1999†

1 Introduction

DiSOM [3, 4, 2] is a distributed shared memory system that offers users an atomic collection of memory cells provided they satisfy certain well-formedness conditions. This report proves the correctness of DiSOM.

The system partitions memory into a set of objects and implicitly associates a read-write lock with each object. Users synchronize accesses to these objects explicitly executing synchronization operations on the associated locks. DiSOM's distributed read-write lock implementation guarantees progress and the usual read-write lock exclusion conditions.

DiSOM guarantees an atomic view of memory provided: (1) all write accesses to an object's cells occur when the object's lock is acquired for writing; and (2) read accesses occur only when the lock is acquired for reading or writing. This model is similar to entry consistency [1] but it is simpler and provides an atomic memory instead of a sequentially consistent one.

2 Interface

DiSOM's memory consistency protocol is implemented by an asynchronous network system D with reliable FIFO channels. We assume there are n users $U_1, ..., U_n$ accessing the distributed shared memory and that D has one process D_i corresponding to each user U_i. We call the composition of the user automata U. Informally, U_i communicates with D_i using *data access* and *synchronization* operations. Data access operations read and write shared memory cells, whereas synchronization operations acquire and release the read-write lock associated with an object. This section defines the actions used for communication between I/O automata U_i and D_i formally.

It starts by introducing some preliminary definitions. Let $\mathcal{C}$ be the set of shared memory cells and $\mathcal{V}$ be the set of their values. A shared object is defined to be a subset of shared memory cells. Let $\mathcal{O}$ be the set of shared objects. We assume that the sets of cells corresponding to each object in $\mathcal{O}$ are pairwise disjoint.

D_i communicates with U_i using the following interface actions (in this definition $v \in \mathcal{V}$, $c \in \mathcal{C}$ and $o \in \mathcal{O}$):

[*] This work was supported by a Praxis XXI fellowship from JNICT.

[†] Originally written in the Fall of 1995

Input:
(* data access input actions *)
request-read$(c)_i$
request-write$(c, v)_i$

(* synchronization input actions *)
request-acq-read$(o)_i$
request-rel-read$(o)_i$
request-acq-write$(o)_i$
request-rel-write$(o)_i$

Output:
(* data access output actions *)
reply-read$(c, v)_i$
reply-write$(c)_i$

(* synchronization output actions *)
reply-acq-read$(o)_i$
reply-rel-read$(o)_i$
reply-acq-write$(o)_i$
reply-rel-write$(o)_i$

We define *corresponding* request and reply actions to be actions with the same type (e.g request-acq-write and reply-acq-write) and the same cell or object (depending on whether they are data access actions or synchronization actions). An *operation* is a pair of consecutive corresponding request and reply actions in a trace.

3 Correctness Conditions

This section defines correctness conditions for system $D \times U$. It starts by defining conditions on the user automata. We define a sequence β of actions from the interface above to be well-formed for user U_i if it satisfies the following conditions:

1. (All operations are blocking) $\beta|i$ starts with a request action and every request action is immediately followed by a corresponding reply action (except possibly for the last request action in the sequence).

2. (Synchronization operations are well formed) For any $o \in \mathcal{O}$, let γ be the subsequence obtained from $\beta|i$ by removing all the data access actions and all the synchronization actions involving objects other than o. Then γ is a sequence starting with either a (request-acq-write$_i$, reply-acq-write$_i$) or a (request-acq-read$_i$, reply-acq-read$_i$) operations; such that every occurrence of (request-acq-write$_i$, reply-acq-write$_i$) is immediately followed by a (request-rel-write$_i$, reply-rel-write$_i$) operation; and every occurrence of (request-acq-read$_i$, reply-acq-read$_i$) is immediately followed by a (request-rel-read$_i$, reply-rel-read$_i$) operation.

We say that U satisfies the well-formedness condition if each U_i preserves the trace property defined by the set of sequences that are well-formed for U_i (according to the definition above).

Given a well-formed sequence of actions β, an object $o \in \mathcal{O}$ and a user U_i, we say that U_i is:

- in its remainder region for o, $R(o)$, in between any reply-rel-write$_i$ or reply-rel-read$_i$ actions with argument o and the following request-acq-write$_i$ or request-acq-read$_i$ actions with argument o.
- in its write trying region for o, $T(o)_w$, in between any request-acq-write$_i$ action with argument o and the corresponding reply action.
- in its write critical region for o, $C(o)_w$, in between any reply-acq-write$_i$ action with argument o and the following request-rel-write$_i$ with argument o.
- in its write exit region for o, $E(o)_w$, in between any request-rel-write$_i$ with argument o and the corresponding reply action.

$T(o)_r, C(o)_r$ and $E(o)_r$ can be defined in a similar way. With these region definitions we can define another well-formedness condition.

Synchronized data accesses: We say that a sequence β of actions from the interface satisfies the synchronized data accesses condition for user U_i if the following holds. For any $o \in \mathcal{O}$, let γ be the subsequence obtained from $\beta|i$ by removing all the data access actions involving a cell d such that $d \notin o$ and all the synchronization actions involving objects other than o. Then γ must satisfy the following conditions, for any applicable $v \in V$ and any $c \in o$:

1. A request-write$(c, v)_i$ action occurs only if U_i is in region $C(o)_w$.
2. A request-read$(c)_i$ action occurs only if U_i is either in region $C(o)_r$ or in region $C(o)_w$.

Intuitively, the last well-formedness condition restricts the contexts in which users are allowed to write or read from shared memory: (1) all write accesses to an object's cells must occur when the object's lock is acquired for writing; and (2) read accesses can only occur when the lock is acquired for reading or writing. We say that U satisfies the synchronized data accesses condition if each U_i preserves the trace property defined by the set of sequences that satisfy the synchronized data accesses condition for U_i (according to the definition above).

We can now define correctness conditions for the entire system $D \times U$.

Well-formedness: In any trace of $D \times U$, the subsequence describing the interaction between U_i and D_i is well-formed for U_i.

Exclusion: For any object o and any reachable system state of $D \times U$, if some U_i is in $C(o)_w$ then no other user is in $C(o)_w$ or $C(o)_r$.

The last condition is the usual exclusion condition for read-write locks. It says that: (1) if user U_i has object o acquired for writing then no other user can have o acquired for writing or reading; and (2) if user U_i has object o acquired for reading then no other user can have o acquired for writing.

Progress: At any point in a fair execution of $D \times U$ and for any object $o \in O$:

1. (Progress for the trying regions) If at least one user is in $T(o)_w$ (or $T(o)_r$) and no user is in $C(o)_w$ or $C(o)_r$, then at some later point some user enters $C(o)_w$ or $C(o)_r$.

2. (Progress for the exit regions) If at least one user is in $E(o)_w$ (or $E(o)_r$) then at some later point some user enters $R(o)$.

We define the underlying variable type $\mathcal{T}$ of the shared memory implemented by D as follows. The set of values of $\mathcal{T}$ is $\mathcal{V}^{|\mathcal{C}|}$, i.e. each state of a variable of type $\mathcal{T}$ is a tuple of values of $\mathcal{V}$ and each component in the tuple corresponds to a cell in $\mathcal{C}$. Given an element $w \in \mathcal{V}^{|\mathcal{C}|}$, we define $w|c$ to be the component of w corresponding to cell c. The initial value of type $\mathcal{T}$ is $(v_o, ..., v_o)$. The set of invocations of $\mathcal{T}$ is $\mathcal{I} = \{\text{request-read}(c) : \forall c \in \mathcal{C}\} \bigcup \{\text{request-write}(c, v) : \forall c \in \mathcal{C}, v \in \mathcal{V}\}$ and the set of responses is $\mathcal{R} = \{\text{reply-read}(c, v) : \forall c \in \mathcal{C}, v \in \mathcal{V}\} \bigcup \{\text{reply-write}(c) : \forall c \in \mathcal{C}\}$. The function $f : \mathcal{I} \times \mathcal{V}^{|\mathcal{C}|} \rightarrow \mathcal{R} \times \mathcal{V}^{|\mathcal{C}|}$ that defines $\mathcal{T}$'s behavior is defined as follows. Let w be an arbitrary element of $\mathcal{V}^{|\mathcal{C}|}$, then $f(\text{request-read}(c), w) = (\text{reply-read}(c, w|c), w)$ and $f(\text{request-write}(c, u), w) = (\text{reply-write}(c), w')$, where $w'|a = w|a$, $\forall a \in \mathcal{C} - \{c\}$ and $w'|c = u$.

Atomicity: Let β be any well formed trace of $D \times U$ and γ be the subsequence obtained from β by removing all actions except data access actions. Then for each operation π in γ, it is possible to insert a serialization point $*_\pi$ between the request action of π and the reply action of π (if π is missing a reply action it must be possible to pick an appropriate reply action and insert $*_\pi$ after the occurrence of π's request); such that the sequence γ', obtained from γ by moving the request and reply actions of each operation π to $*_\pi$ in this order, is a trace of $\mathcal{T}$.

We say that D is correct, if for every collection of users U that satisfies the well-formedness and synchronized data accesses conditions, $D \times U$ satisfies the well-formedness, exclusion, progress and atomicity conditions.

4 Algorithm

DiSOM's memory consistency protocol is implemented by an asynchronous send/receive system D. This section describes the algorithm implementing each process D_i in D. Let $\mathcal{P} = \{1, ..., n\}$ be the set of processes in D. We assume there is one universal reliable FIFO channel $C_{i,j}$ connecting process D_i to D_j, for each $j \neq i$ in $\mathcal{P}$. The first sub-section presents an informal description of the algorithm and the second presents a formal definition of each automaton D_i.

4.1 Informal Description

The distributed read-write lock algorithm associates two types of tokens with each lock, the *write token* and the *read token*. For an acq-write operation on an object o to complete, the issuing process must hold o's write token. An acq-read operation will only complete if the process holds a read token. The algorithm ensures that: (1) if one process holds a write token for an object o then no other processes hold tokens for o; and (2) if one process holds a read token for o then no process holds a write token for o. This invariant ensures the exclusion condition.

Each lock has an associated *owner*. The owner is either the process holding the write token or the last process to hold a write token. Ownership is dynamic, to keep track of the owner, each process maintains a forwarding pointer, $probOwner(o)$, which points to the process it believes is the lock owner. The algorithm ensures that the owner can always be reached following the forwarding pointer chain. The owner process remembers which processes have obtained read tokens using the $tokenSet(o)$ variable. The $requests(o)$ queue is used to remember token transfer requests until they can be serviced.

Processes cache tokens and can re-acquire locks with no communication as long as their token is not invalidated. When a process executes request-acq-write or request-acq-read and does not hold the needed token, a message is sent requesting the token. The message is sent along the forwarding pointer chain. A token can only be obtained from the owner. Therefore, forwarding stops when the message reaches the owner. As an optimization, if the request message reaches a process requesting a write token for the same object, the request is queued at that process.

When a process receives a request for a write token that it holds, it waits until the lock is released by its user and then it replies with a message transferring the write token, the $tokenSet(o)$ and the rest of the $requests(o)$ queue to the requester. The owner then sets the forwarding pointer to point to the requester. When the requester receives the reply it prepends the queue of requests to its own. Then it sends messages to all the processes in the received $tokenSet(o)$, invalidating their read tokens, and waits for the replies.

When the owner receives a request for a read token and has a read token, the requesting process is inserted in $tokenSet(o)$ and a reply is sent to it. The reply includes the read token and the owner identity. Thus all readers have their forwarding pointers set correctly. If the owner receives a read token request and it holds a write token, it proceeds as above but first converts its write token into a read token.

DiSOM ensures the atomicity condition using an update protocol that piggy-backs the values of the cells in an object in the token transfer messages.

4.2 Formal definition

The messages used by the algorithm are elements of set $\mathcal{M}$ defined next.

$$\begin{aligned}\mathcal{M} = {} & (\{reqWrite, reqRead, reqInv, repInv\} \times \mathcal{P} \times \mathcal{O}) \bigcup \\ & (\{repWrite\} \times \mathcal{P} \times \mathcal{O} \times 2^{\mathcal{C}\times\mathcal{V}} \times 2^{\mathcal{P}} \times \mathcal{Q}) \bigcup \\ & (\{repRead\} \times \mathcal{P} \times \mathcal{O} \times 2^{\mathcal{C}\times\mathcal{V}})\end{aligned}$$

In this definition $\mathcal{O}$,$\mathcal{C}$ and $\mathcal{V}$ are the sets defined in Section 2, and $\mathcal{Q}$ is the set of all FIFO queues of elements of $\{(t,p) : t \in \{reqRead, reqWrite, reqInv\}, p \in \mathcal{P}\}$. We also define a function $home : \mathcal{O} \rightarrow \mathcal{P}$, which maps each object o to a fixed home process $home(o)$. The I/O automaton for process D_i has the following signature (in this definition $v \in \mathcal{V}$, $c \in \mathcal{C}$, $o \in \mathcal{O}$, $m \in \mathcal{M}$):

Input:
request-read$(c)_i$
request-write$(c, v)_i$
request-acq-read$(o)_i$
request-rel-read$(o)_i$
request-acq-write$(o)_i$
request-rel-write$(o)_i$
receive$(m)_{j,i}$

Output:
reply-read$(c, v)_i$
reply-write$(c)_i$
reply-acq-read$(o)_i$
reply-rel-read$(o)_i$
reply-acq-write$(o)_i$
reply-rel-write$(o)_i$

send$(m)_{i,j}$

Internal:
reply-invalidate$(o)_i$

Each process maintains a *status* variable which keeps track of its computation status. This variable takes values in the set $\mathcal{S}$ defined next.

$$\begin{aligned}\text{Let } \mathcal{S} = {} & (\{repRead\} \times \mathcal{C} \times \mathcal{V}) \bigcup \\ & (\{repWrite\} \times \mathcal{C}) \bigcup \\ & (\{repAcqRead, repRelRead, repAcqWrite, repRelWrite, repInvalidate, \\ & reqAcqRead, reqAcqWrite\} \times \mathcal{O}) \bigcup \{idle\}\end{aligned}$$

D_i has the following state components:
$status \in \mathcal{S}$, initially *idle*
For each $o \in \mathcal{O}$,
  $token(o) \in \{read, write, none\}$, initially *write* in $home(o)$ and *none* elsewhere
  $held(o) \in \{read, write, false\}$, initially $false$
  $requests(o) \in \mathcal{Q}$, initially *empty*
  $tokenSet(o) \in 2^{\mathcal{P}}$, initially *empty*
  $probOwner(o) \in \mathcal{P}$, initially $home(o)$
For each $c \in \mathcal{C}$,
  $value(c) \in V$, initially v_0
For each j in $\mathcal{P} - \{i\}$
  $out(j)$ is a queue of elements of $\mathcal{M}$, initially *empty*

D_i has the following transitions:

```
request-read$(c)_i$
    Effect:
        $status := (repRead, c, value(c))$

reply-read$(c, v)_i$
    Precondition:
        $status = (repRead, c, v)$
    Effect:
        $status := idle$

request-write$(c, v)_i$
    Effect:
        $value(c) := v$
        $status := (repWrite, c)$

reply-write$(c)_i$
    Precondition:
        $status = (repWrite, c)$
    Effect:
        $status := idle$

request-acq-read$(o)_i$
    Effect:
        $held(o) := read$
```

```
      if $(token(o) \neq none)$ then
        $token(o) := read$
        $status := (repAcqRead, o)$
      else
        append $(reqRead, i, o)$ to $out(probOwner(o))$
        $status := (reqAcqRead, o)$

receive$(repRead, i, o, u)_{j,i}$
  Effect:
      if $(status = (reqAcqRead, o))$ then
        for each $(c, v) \in u$ do
            $value(c) := v$
        $probOwner(o) := j$
        $token(o) := read$
        $status := (repAcqRead, o)$

reply-acq-read$(o)_i$
  Precondition:
      $status = (repAcqRead, o)$
  Effect:
      $status := idle$

receive$(reqRead, p, o)_{j,i}$
  Effect:
      if $(probOwner(o) = i \wedge held(o) \neq write)$ then
        $tokenSet(o) := tokenSet(o) \cup \{p\}$
        $token(o) := read$
        $u := \{\}$
        for each $c \in o$ do
            $u := u \cup (c, value(c))$
        append $(repRead, p, o, u)$ to $out(p)$
      else if $(probOwner(o) \neq i \wedge held(o) \neq write)$ then
        append $(reqRead, p, o)$ to $out(probOwner(o))$
      else
        append $(reqRead, p)$ to $requests(o)$

request-rel-read$(o)_i$
  Effect:
      $held(o) := false$
      $status := (repRelRead, o)$
      if $(requests(o)$ is not $empty)$ then
        $(t, p) :=$ first element of $requests(o)$
        remove first element of $requests(o)$
        if $(probOwner(o) = i)$ then
          $token(o) := none$
          $u := \{\}$
          for each $c \in o$ do
              $u := u \cup (c, value(c))$
          append $(repWrite, p, o, u, tokenSet(o), requests(o))$ to $out(p)$
          $tokenSet(o) := \{\}$
          $probOwner(o) := p$
        else
          $token(o) := none$
```

```
            $probOwner(o) := p$
            append $(repInv, p, o)$ to $out(p)$

reply-rel-read$(o)_i$
  Precondition:
      $status = (repRelRead, o)$
  Effect:
      $status := idle$

request-acq-write$(o)_i$
  Effect:
      $held(o) := write$
      if $(probOwner(o) = i)$ then
        $status := (repInvalidate, o)$
        for each $p \in tokenSet(o)$ do
            append $(reqInv, i, o)$ to $out(p)$
      else
        append $(reqWrite, i, o)$ to $out(probOwner(o))$
        $status := (reqAcqWrite, o)$

receive$(repWrite, i, o, u, t, r)_{j,i}$
  Effect:
      if $(status = (reqAcqWrite, o))$ then
        for each $(c, v) \in u$ do
            $value(c) := v$
        $probOwner(o) := i$
        $tokenSet(o) := t - \{i\}$
        prepend $r$ to $requests(o)$
        $status := (repInvalidate, o)$
        for each $p \in tokenSet(o)$ do
            append $(reqInv, i, o)$ to $out(p)$

reply-acq-write$(o)_i$
  Precondition:
      $status = (repAcqWrite, o)$
  Effect:
      $status := idle$

receive$(reqWrite, p, o)_{j,i}$
  Effect:
      if $(probOwner(o) = i \wedge held(o) = false)$ then
        $token(o) := none$
        $u := \{\}$
        for each $c \in o$ do
            $u := u \cup \{(c, value(c))\}$
        append $(repWrite, p, o, u, tokenSet(o), requests(o))$ to $out(p)$
        $tokenSet(o) := \{\}$
        $probOwner(o) := p$
      else if $(probOwner(o) \neq i \wedge held(o) \neq write)$ then
        append $(reqWrite, p, o)$ to $out(probOwner(o))$
      else
        append $(reqWrite, p)$ to $requests(o)$
```

```
request-rel-write$(o)_i$
    Effect:
        $held(o) := false$
        $status := (repRelWrite, o)$
        $u := \{\}$
        for each $c \in o$ do
            $u := u \cup (c, value(c))$
        while ($requests(o)$ is not $empty$) do
                $(t, p) :=$ first element of $requests(o)$
                remove first element of $requests(o)$
                if $(t = reqWrite)$ then
                    $token(o) := none$
                    append $(repWrite, p, o, u, tokenSet(o), requests(o))$ to $out(p)$
                    $tokenSet(o) := \{\}$
                    $probOwner(o) := p$
                    break
                else
                    $token(o) := read$
                    $tokenSet(o) := tokenSet(o) \cup \{p\}$
                    append $(repRead, p, o, u)$ to $out(p)$

reply-rel-write$(o)_i$
    Precondition:
        $status = (repRelWrite, o)$
    Effect:
        $status := idle$

receive$(reqInv, j, o)_{j,i}$
    Effect:
        if $(held(o) \neq read)$ then
            $token(o) := none$
            $probOwner(o) := j$
            append $(repInv, j, o)$ to $out(j)$
        else
            append $(reqInv, j)$ to $requests(o)$

receive$(repInv, i, o)_{j,i}$
    Effect:
        if $(status = (repInvalidate, o))$ then
            $tokenSet(o) := tokenSet(o) - \{j\}$

reply-invalidate$(o)_i$
    Precondition:
        $status = (repInvalidate, o)$
        $tokenSet(o) = \{\}$
    Effect:
        $token(o) := write$
        $status := (repAcqWrite, o)$

send$(m)_{i,j}$
    Precondition:
        $m$ is first element of $out(j)$
    Effect:
        remove first element of $out(j)$
```

D_i has the following tasks:
For each $j \in \mathcal{P}$,
$\{\text{send}(m)_{i,j} : \forall m \in \mathcal{M}\}$
and another task for all the other locally controlled actions.

5 Proof of correctness

We start by proving the well-formedness condition holds for any system $D \times U$ such that U satisfies well-formedness. Then we prove that exclusion and progress hold in the same conditions. The last subsection shows that D provides an atomic memory to any set of users U that satisfy both the well-formedness and the synchronized data accesses conditions.

5.1 Well-formedness

Theorem 1: If U satisfies the well-formedness condition then $D \times U$ satisfies the well-formedness condition.

Proof: The *status* variable of each process i ensures that exactly one output reply action occurs in response to a request input action and that the two actions correspond. This combined with the fact that U satisfies well-formedness implies that $D \times U$ also satisfies well-formedness.

5.2 Exclusion

This section proves that the exclusion condition holds for any system $D \times U$ that satisfies well-formedness. The proof is based on the following auxiliary lemmas.

Lemma 1: If $D \times U$ satisfies the well-formedness condition then, in any state of $D \times U$, any $o \in \mathcal{O}$, any $i \in \mathcal{P}$, any $j \in \mathcal{P} - \{i\}$, and appropriate values of u, t and r, the following conditions are satisfied:

1. If $probOwner(o)_i = i$ then $probOwner(o)_j \neq j$
2. If $token(o)_i = write$ then $probOwner(o)_i = i \ \wedge \ tokenSet(o)_i = \{\}$
3. If $status_i = (repInvalidate, o)$ then $probOwner(o)_i = i \ \wedge \ held(o)_i = write$
4. If $status_i = (reqAcqWrite, o)$ then $held(o)_i = write$
5. If $status_i = (repAcqWrite, o)$ then $token(o)_i = write \ \wedge \ held(o)_i = write$
6. If $(repWrite, j, o, u, t, r)$ is in $C_{i,j}$ or in $out(j)_i$ then $\forall p \in \mathcal{P} : \ probOwner(o)_p \neq p$
7. If $(reqInv, i, o)$ is in $C_{i,j}$ or in $out(j)_i$ then $status_i = (repInvalidate, o)$
8. If $probOwner(o)_i = i$ then $token(o)_i \neq none \ \vee \ status_i = (repInvalidate, o)$

Proof: By induction on the length of an execution. Fix any object o.
Base case: The initializations ensure that $\forall i \in \mathcal{P} : \mathit{probOwner}(o)_i = \mathit{home}(o)$, therefore (1) holds initially. The initializations also ensure that $\forall i \in \mathcal{P} : \mathit{tokenSet}(o)_i = \{\}$, $\mathit{token}(o)_{\mathit{home}(o)} = \mathit{write}$ and $\forall i \neq \mathit{home}(o) : \mathit{token}(o)_i = \mathit{none}$. Therefore (2) and (8) are also satisfied in the base case. (3), (4) and (5) are initially vacuously true because the *status* variables are initialized to *idle*. Since all channels are and all *out* variables are initially *empty* (6) and (7) are vacuously true.
Inductive step: Assume the lemma holds for every state of any execution α of length at most l. We will show it holds for any one step extension α_1 of α.

The only way for (1) to be violated in α_1 is if in the last state in α there is a process i such that $\mathit{probOwner}(o)_i = i$ and some action of another process j is executed that sets $\mathit{probOwner}(o)_j = j$. The only actions that can do this are actions of the form receive$(\mathit{repWrite}, j, o, u, t, r)_{p,j}$ (for some p different from j). An action of this form can only occur if the appropriate message is at the head of channel $C_{p,j}$ in the last state in α. Therefore, from the inductive hypothesis of (6), in this state, $\forall p \in \mathcal{P} : \mathit{probOwner}(o)_p \neq p$. This is enough to conclude that actions of this form can not violate (1).

To prove (2), (3), (4), (5), (6), (7) and (8), we consider two cases (a) the antecedent is false in the last state of α and (b) the antecedent is true in this state.

In case (a) the only actions that can violate (2) are those that set $\mathit{token}(o)_i = \mathit{write}$, i.e. actions of the form reply-invalidate$(o)_i$. Actions of this type are only enabled if $\mathit{status}_i = (\mathit{repInvalidate}, o)$ therefore we can use the inductive hypothesis of (3) to conclude that if an action of this type occurs then $\mathit{probOwner}(o)_i = i \wedge \mathit{tokenSet}(o)_i = \{\}$ in the last state of α, which shows that (2) can not be violated in this case.

In case (b), only actions that set $\mathit{probOwner}(o)_i \neq i$ or set $\mathit{tokenSet}(o) \neq \{\}$, and leave $\mathit{token}(o)_i = \mathit{write}$ can violate (2). The inductive hypothesis for (2) implies that $\mathit{probOwner}(o)_i = i$ in the last state of α. Therefore, only actions of the form receive$(\mathit{repWrite}, i, o, u, t, r)_{j,i}$ can violate (2) in this case. However, if one of these actions is enabled we can apply the inductive hypothesis of (6) and (2) to conclude that $\mathit{token}(o)_i \neq \mathit{write}$ in the last state of α, which violates our assumption for case (b). Therefore, (2) is satisfied in α_1.

In case (a) only actions that set $\mathit{status}_i = (\mathit{repInvalidate}, o)$ can violate (3), i.e. actions of the form request-acq-write$(o)_i$ and receive$(\mathit{repWrite}, i, o, u, t, r)_{j,i}$. The first type of actions can not violate (3), because they always set $\mathit{held}(o)_i = \mathit{write}$ and only set $\mathit{status} = (\mathit{repInvalidate}, o)$ if $\mathit{probOwner}(o)_i = i$. The second type of actions, only modify the state of i if $\mathit{status}_i = (\mathit{reqAcqWrite}, o)$ and when they do so they set $\mathit{probOwner}(o)_i = i$. Therefore, since these actions do not modify $\mathit{held}(o)_i$, we can use the inductive hypothesis of (4) to conclude that they do not violate (3) in this case.

In case (b) only actions that set $\mathit{probOwner}(o)_i \neq i$ or $\mathit{held}(o)_i \neq \mathit{write}$ and do not modify status_i can violate (3), i.e. receive$(\mathit{reqWrite}, p, o)_{j,i}$ and receive$(\mathit{reqInv}, j, o)_{j,i}$. In case (b), from the inductive hypothesis for (4), $\mathit{held}(o)_i = \mathit{write}$ in the last state of α, therefore the first type of actions does not modify $\mathit{probOwner}(o)_i$. Since, these actions never modify $\mathit{held}(o)_i$, we conclude that they preserve the invariant in this case. For the last type of actions, the conditions of case (b) and the inductive hypothesis for (3) imply that $\mathit{probOwner}(o)_i = i$ in the last state of α. Therefore, we can use the inductive hypothesis of (1) to conclude that no other $j \neq i$ can have $\mathit{probOwner}(o)_j = j$. Combining this with the inductive hypothesis of (3) and (7) we conclude that actions of the form receive$(\mathit{reqInv}, j, o)_{j,i}$ can not violate (3) in case (b).

In what concerns (4), in case (a) only actions of the form request-acq-write$(o)_i$ set $\mathit{status} = (\mathit{reqAcqWrite}, o)$ but they also set $\mathit{held}(o)_i = \mathit{write}$. Therefore, (4) can not be violated in this case. In case (b) there is no action that sets $\mathit{held}(o)_i \neq \mathit{write}$ without modifying status_i. Therefore,

(4) can not also be violated in this case.

For (5), in case (a), the only actions that can violate the invariant are those that set $status_i = (repAcqWrite, o)$, i.e. reply-invalidate$(o)_i$. This action is only enabled if $status_i = (repInvalidate, o)$. Therefore, the inductive hypothesis of (3) implies that $held(o)_i = write$ just before this action occurs. Since this action does not modify $held(o)_i$ and always sets $token(o)_i = write$, we conclude that it preserves (5) in this case.

In case (b), only actions that do not modify status and set $token(o)_i \neq write$ or $held(o)_i \neq write$ can violate (5), i.e. actions of the form receive$(reqRead, p, o)_{j,i}$, receive$(reqWrite, p, o)_{j,i}$, receive$(reqInv, j, o)$. Using the hypothesis of case (b) and the inductive hypothesis of (5), we conclude that the first two types of actions can not violate the invariant because $held(o)_i = write$ in the last state of α. The third type of action can not violate the invariant because the inductive hypothesis of (5), (2) and (7) imply that i can not receive a $reqInv$ message in this case.

Only actions of the forms request-rel-read$(o)_i$, receive$(reqWrite, p, o)_{j,i}$, and request-rel-write$(o)_i$, can violate (6) in case (a). Actions of the first two types only append a $repWrite$ message to an out variable if $probOwner(o)_i = i$ in the last state of α and when they do so they set $probOwner(o)_i \neq i$. Therefore, we can apply the inductive hypothesis of (1) to conclude that they preserve the invariant in case (a). Well-formedness and the inductive hypothesis of (2) imply that the last type of actions can only occur when $probOwner(o)_i = i$ in the last state of α. Therefore, we can use the same argument to conclude that these actions also preserve the invariant in case (a).

In case (b), we note that for a given object o there can be at most one $(repWrite, j, o, u, t, r)$ message in a channel or out variable at any point in an execution. This is true because the actions that send such messages only do so if $probOwner(o)_i = i$, and the inductive hypothesis of (6) implies that if there is such a message no process p has $probOwner(o)_p = p$. Therefore, no process can send a message of this type while there is one in transit. The only actions that can violate (6) in this case are those of the type receive$(repWrite, i, o, u, t, r)_{j,i}$. However, using the invariant just proven we conclude that these actions preserve (6) in case (b) because they make the antecedent false.

In case (a), the only actions that can violate (7) are actions that send $reqInv$ messages, i.e. actions of the form request-acq-write$(o)_i$ or receive$(repWrite, i, o, u, t, r)_{j,i}$. Actions of both types set $status_i = (repInvalidate, o)$ thus they preserve the invariant in this case.

In case (b), all actions that set $status_i \neq (repInvalidate, o)$ can potentially violate (7). However, well-formedness prevents all request input actions from occurring. The inductive hypothesis of (7) together with the conditions for case (b) ensure that $status_i = (repInvalidate, o)$ in the last state of α. Therefore, no reply output action is enabled. Therefore, the only actions that can potentially violate (7) in this case are of the form reply-invalidate$(o)_i$. But these actions are only enabled if $tokenSet(o)_i = \{\}$ which in turn implies that there are no $repInv$ messages in transit. Therefore, (7) holds in both cases.

For (8), in case (a) only actions that set $probOwner(o)_i = i$ can violate the invariant, i.e. receive$(repWrite, i, o, u, t, r)_{j,i}$. However, these actions set $status_i = (repInvalidate, o)$ and therefore do not violate (8).

In case (b), if $status_i = (repInvalidate, o)$ well-formedness prevents the occurrence of any request action and of all output reply actions except for reply-acq-write$(o)_i$ (which is not enabled in this case). Therefore, the only actions that can set $status_i \neq (repInvalidate, o)$ are actions of the form reply-invalidate$(o)_i$, but these actions also set $token(o)_i = write$, thus they preserve (8). In case (b), if $status_i \neq (repInvalidate, o)$ in the last state of α then $token(o)_i \neq none$ in this state. In this case, the only actions that can violate the (8) are those that set $token(o)_i = none$, but all of these actions set $probOwner(o)_i \neq i$ and doing so preserve (8).

Lemma 2: If $D \times U$ satisfies the well-formedness condition then, in any state of $D \times U$, for any $o \in \mathcal{O}$, any $i, j, p \in \mathcal{P}$ such that $i \neq j \ \wedge \ i \neq p \ \wedge \ j \neq p$, and for appropriate values of u, v, t and r the following conditions are satisfied.

1. If $status_i = (reqAcqRead, o)$ then $held(o)_i = read \ \wedge \ token(o)_i = none$
2. If $status_i = (repAcqRead, o)$ then $held(o)_i = read \ \wedge \ token(o)_i = read$
3. If there is a $(repRead, j, o, u)$ in $C_{i,j}$ or $out(j)_i$ then (i) $probOwner(o)_i = i \wedge j \in tokenSet(o)_i$ or (ii) $probOwner(o)_i = p \wedge status_p = (repInvalidate, o) \wedge j \in tokenSet(o)_p$ or (iii) there is one $(repWrite, p, o, v, t, r)$ message in $out(p)_i$ or $C_{i,p}$ such that $j \in t$ and $probOwner(o)_i = p$.
4. If $token(o)_j = read$ then (i) $probOwner(o)_j = j$ or (ii) $probOwner(o)_i = i \wedge j \in tokenSet(o)_i$ or (iii) there is one $(repWrite, p, o, v, t, r)$ message in $out(p)_i$ or $C_{i,p}$ such that $j \in t$.
5. If $probOwner(o)_i = i \wedge token(o)_j = read$ then $j \in tokenSet(o)_i$.
6. If there is a $(repInv, j, o)$ message in $C_{i,j}$ or $out(j)_i$ then $token(o)_i = none \ \wedge \ status_j = (repInvalidate, o) \wedge$ there is no $(repRead, i, o, u)$ in $C_{q,i}$ or $out(i)_q$ for any $q \in \mathcal{P}$.

Proof: We start by noting that (4) and parts (1) and (6) of lemma 1 imply (5). We prove the rest of the lemma by induction on the length of an execution. Fix any object o.

Base case: The initializations ensure that $status_i$ is initially *idle*. Therefore, (1) and (2) are vacuously true in the base case. The initializations also ensure that $token(o)_i \neq read$ for any $i \in \mathcal{P}$ and that the channels and *out* variables are *empty*. Therefore, (3), (4) and (6) are also true in the base case.

Inductive step: Assume the lemma holds for every state of any execution α of length at most l. We will show it holds for any one step extension α_1 of α.

We consider two cases in the proofs, (a) the antecedent is false in the last state of α and (b) the antecedent is true in this state.

For (1), in case (a) the only actions that can potentially violate the invariant are those that set $status_i = (reqAcqRead, o)$, i.e. request-acq-read$(o)_i$. However, these actions always set $held(o)_i = read$ and only set $status_i = (reqAcqRead, o)$ if $token(o)_i = none$. Therefore, they do not violate (1).

In case (b), there are no actions that set $held(o)_i \neq read$ without modifying $status_i$. The only actions that can set $token(o)_i \neq none$ without modifying $status_i$ are actions of the form receive$(reqRead, p, o)_{j,i}$, but part (8) of lemma 1 together with the conditions for case (b) implies that $probOwner(o)_i \neq i$ in the last state of α and thus these actions can not modify $token(o)_i$ and the invariant holds.

In what concerns (2), in case (a), the only actions that can potentially violate the invariant are those that set $status_i = (repAcqRead, o)$, i.e. request-acq-read$(o)_i$ and receive$(repRead, i, o, u)_{j,i}$. The first type of action always sets $held(o)_i = read$ and when it sets $status_i = (repAcqRead, o)$ it also sets $token(o)_i = read$. Therefore, (2) is preserved by this type of actions in this case. The second type of actions only modifies the state of i, if $status_i = (reqAcqRead, o)$ in the last state of α and always sets $token(o)_i = read$ in this case. Using the inductive hypothesis of (1) we conclude that this type of actions also preserves (2) in case (a). In case (b), only actions that do not modify $status_i$ and set $held(o)_i \neq read$ or $token(o)_i \neq read$ can potentially violate (2), i.e. receive$(reqWrite, p, o)_{j,i}$ and receive$(reqInv, j, o)_{j,i}$. In case (b) the inductive hypothesis of (2) implies that $held(o)_i = read$, therefore actions of these types can not violate (2) in case (b).

For (3), in case (a), only actions that put a $repRead$ message in $out(j)_i$ can violate the invariant, i.e. actions of the forms receive$(reqRead, j, o)_{q,i}$ or request-rel-write$(o)_i$. The first type of action only sends a $repRead$ message if $probOwner(o)_i = i$, it does not modify $probOwner(o)_i$ and it inserts j in $tokenSet(o)_i$. Therefore, this type of action preserves the invariant. Well-formedness and part (2) of lemma 1 imply that the second type of actions can only occur if $probOwner(o)_i = i$. Also note that these actions insert the indices of the processes to which $repRead$ messages are sent in $tokenSet(o)_i$. We distinguish two cases (a.1) $requests(o)_i$ contains a $reqWrite$ request or (a.2) it does not. In case (a.1) let $(reqWrite, p)$ be the first write request in $requests(o)_i$, then (iii) will hold after the action executes. In case (a.2), (i) will hold after the action executes. Therefore, actions of this type also preserve (3) in case (a).

In case (b) for (3), we note that only one of (i) to (iii) can be true at any point in an execution. This is so because of lemma 1 parts (1), (3) and (6). We consider 3 cases (b.1) to (b.3) corresponding to exactly one of (i) to (iii) being true in the last state of α. In case (b.1), only actions that set $probOwner(o)_i \neq i$ or remove j from $tokenSet(o)_i$ can potentially violate (3), i.e. receive$(repRead, i, o, u)_{j,i}$, request-rel-read$(o)_i$, receive$(reqWrite, p, o)_{j,i}$, request-rel-write$(o)_i$, receive$(reqInv, j, o)_{j,i}$ and receive$(repInv, i, o)_{j,i}$. Actions of the form receive$(repRead, i, o, u)_{j,i}$ only have effects if $status_i = (reqAcqRead, o)$. Since in this case $probOwner(o)_i = i$ in the last state of α, part (8) of lemma 1 and (1) imply that these actions can not violate the invariant in this case. Actions of the forms request-rel-read$(o)_i$, receive$(reqWrite, p, o)_{j,i}$ or request-rel-write$(o)_i$ do not violate (3) in this case, because when they modify $probOwner(o)_i$ they ensure that (iii) is true just after they execute. Actions of the form receive$(reqInv, j, o)_{j,i}$ can not occur in case (b.1) because of parts (1) and (7) of lemma 1. Finally, receive$(repInv, i, o)_{j,i}$ does not violate (3) in this case because according to the inductive hypothesis of (6) it can not occur in this case. In case (b.2), well-formedness and lemma 1 imply that only receive$(repInv, i, o)_{j,i}$ can violate (3) in this case. However, part (4) of lemma 1 implies that $probOwner(o)_p = p$ in the last state of α, therefore we can apply the inductive hypothesis of (6) to conclude that it can not occur in this case. In case (b.3), only actions that consume the $repWrite$ message can violate (3), but these actions ensure that (ii) is true just after they execute. Therefore, we conclude that (3) holds in all cases.

For (4), in case (a), only actions that set $token(o)_j = read$ can violate the invariant, i.e. request-acq-read$(o)_j$, request-rel-write$(o)_j$, and receive actions corresponding to the arrival of a $reqRead$ or $repRead$ message to j. In case (a), actions of the form request-acq-read$(o)_j$ only set $token(o)_j = read$ if in the last state of α $token(o)_j = write$, which in turn implies (part (2) of lemma 1) that $probOwner(o)_j = j$. Therefore, these actions preserve (4) in case (a). Well-formedness and lemma 1 imply that request-rel-write$(o)_j$ actions can only occur if $probOwner(o)_j = j$ and when these actions set $token(o)_j = read$ they do not modify $probOwner(o)_j$. Therefore, these actions also preserve (4) in this case. A similar reasoning can be used to show that the arrival of a $reqRead$ message at j preserves (4). The last type of actions also preserves (4) because of the inductive hypothesis of (3).

To prove (4) in case (b), we consider 3 cases (b.1) to (b.3) corresponding to exactly one of (i) to (iii) being true in the last state of α (using the same argument as we did for (3)). In case (b.1), (4) can be violated by actions that set $probOwner(o)_j \neq j$, i.e. request-rel-read$(o)_j$, request-rel-write$(o)_j$, and the arrival of a $reqWrite$, $reqInv$, or $repRead$ message at j. All the actions except the last preserve the invariant because they set $token(o)_j = none$ when they modify $probOwner(o)_j$. The last type of actions only modifies the state of j if $status_j = (reqAcqRead, o)$, in the last state of α, but the inductive hypothesis of (1) implies that this can not happen in this case. Therefore, (4) holds in case (b.1). Arguments similar to those used in cases (b.1) and (b.3) for (3), show that (4) also holds in cases (b.2) and (b.3).

For (6), in case (b), the only actions that can potentially violate the invariant are those that append a (repInv, j, o) message to $\mathit{out}(j)_i$, i.e. request-rel-read$(o)_i$ and receive$((\mathit{reqInv}, j, o)_{j,i})$. We note that they both set $\mathit{token}(o)_i = \mathit{none}$ when they send a (repInv, j, o) message. These actions only send a *repInv* message in reply to a *reqInv* message. Part (7) of lemma 1 implies that $\mathit{status}_j = (\mathit{repInvalidate}, o)$ when the *reqInv* message is received and inspecting the code we conclude that status_j retains its value until all the relevant *repInv* replies arrive. Well-formedness implies that there can only be a $(\mathit{repRead}, i, o, u)$ message in transit if $\mathit{status}_i = (\mathit{reqAcqRead}, o)$ and (1) implies that $\mathit{held}(o)_i = \mathit{read}$ in this case. Since, the actions that send (repInv, j, o) messages are either not enabled for this status_i value or do not send the reply message if $\mathit{held}(o)_i = \mathit{read}$, we can conclude that the invariant is preserved in case (a). In case (b), we note that $\mathit{token}(o)_i$ can only become different from *none* if a *repRead* or *repWrite* message for object o is received by i. However, the inductive hypothesis prevents this from happening for *repRead* messages and lemma 1 prevents this from happening for *repWrite* messages. status_j can only become different from $(\mathit{repInvalidate}, o)$ if all *repInv* replies are received. Finally, from lemma 1, if $\mathit{status}_j = (\mathit{repInvalidate}, o)$ then $\mathit{held}(o)_j = \mathit{write} \wedge \mathit{probOwner}(o)_j = j$, therefore no $(\mathit{repRead}, i, o, u)$ will be sent while $\mathit{status}_j = (\mathit{repInvalidate}, o)$.

Theorem 2: If $D \times U$ satisfies the well-formedness condition then it satisfies the exclusion condition.

Proof: For any object $o \in \mathcal{O}$ and any $i \in \mathcal{P}$. From well-formedness and the definition of $C(o)_w$, U_i enters $C(o)_w$ when a (request-acq-write$_i$, reply-acq-write$_i$) operation on o completes and exits this region when a request-rel-write$_i$ action, π, on o occurs. From part (5) of lemma 1, $\mathit{held}(o)_i = \mathit{write}$ and $\mathit{token}(o)_i = \mathit{write}$ just after U_i enters $C(o)_w$. Well-formedness and lemma 1 imply that $\mathit{held}(o)_i = \mathit{write}$ and $\mathit{token}(o)_i = \mathit{write}$ until π. Therefore, we conclude that (a) if U_i is in $C(o)_w$ then $\mathit{held}(o)_i = \mathit{write}$ and $\mathit{token}(o)_i = \mathit{write}$.

A similar argument can be used for $C(o)_r$. U_i enters $C(o)_r$ when a (request-acq-read$_i$, reply-acq-read$_i$) operation on o completes and exits this region when a request-rel-read$_i$ action, θ, on o occurs. From part (2) of lemma 2, $\mathit{held}(o)_i = \mathit{read}$ and $\mathit{token}(o)_i = \mathit{read}$ just after U_i enters $C(o)_r$. Well-formedness and the fact that $\mathit{held}(o)_i = \mathit{read}$ ensure that $\mathit{token}(o)_i$ and $\mathit{held}(o)_i$ retain these values until θ occurs. Therefore, (b) if U_i is in $C(o)_r$ then $\mathit{held}(o)_i = \mathit{read}$ and $\mathit{token}(o)_i = \mathit{read}$.

Assertion (a) and part (2) of lemma 1 imply that (c) if U_i is in $C(o)_w$ then $\mathit{probOwner}(o)_i = i \wedge \mathit{tokenSet}(o)_i = \{\}$. Assertion (c), part (1) of lemma 1 and part (5) of lemma 2 imply that if some U_i is in $C(o)_w$ then no other user is in $C(o)_w$ or $C(o)_r$, as desired.

5.3 Progress

This section shows that if a system $D \times U$ satisfies well-formedness, then $D \times U$ also satisfies progress. We start by proving some auxiliary lemmas.

Lemma 3: In any well-formed fair execution α of $D \times U$, if at some point in α a message m is appended to $\mathit{out}(j)_i$ then eventually a receive$(m)_{i,j}$ occurs.

Proof: Implied by fairness of α.

Lemma 4: In any well-formed fair execution α of $D \times U$, if at some point in α no user is in $C(o)_w$ or $C(o)_r$ and some process i appends a (reqInv, i, o) message to $\mathit{out}(j)_i$ then i eventually receives a (repInv, i, o) message in $C_{j,i}$, or some user enters $C(o)_w$ or $C(o)_r$.

Proof: Fix any object $o \in \mathcal{O}$ and any well-formed fair execution α of $D \times U$. Lemma 3 implies that a receive$(reqInv, i, o)_{i,j}$ occurs at some later point in α. We consider two cases (a) $held(o)_j \neq read$ at this point and (b) $held(o)_j = read$. In the first case the receive action appends a $(repInv, i, o)$ message to $out(i)_j$. Once more lemma 3 implies that eventually a receive$(repInv, i, o)_{j,i}$ occurs and therefore the lemma holds in this case. In case (b), U_j is in $T(o)_r$ or $C(o)_r$. If U_j is in $C(o)_r$ the lemma holds. Therefore, consider the case where U_j is in $T(o)_r$. In this case, if $status_j \neq (reqAcqRead, o)$ then fairness of α implies that U_j will eventually enter $C(o)_r$. Otherwise, since j is receiving an invalidation request, there must be a $(repRead, j, o)$ message in transit. Lemma 3 implies that j will receive a $(repRead, j, o)$ message that will set $status_i = (repAcqRead, o)$. Therefore, by fairness U_j will eventually enter $C(o)_r$ and the lemma also holds in this case.

Lemma 5: If $D \times U$ satisfies well-formedness then in any reachable system state and for any $i \in \mathcal{P}$, the *path* defined by the following sequence of nodes

$$probOwner(o)_i, probOwner(o)_{probOwner(o)_i}, \cdots$$

connects i to a node j such that exactly one of the following holds: (i) $probOwner(o)_j = j$ or (ii) there is a $(repWrite, j, o, u, t, r)$ message in $out(j)_k$ or $C_{k,j}$, and k is the node just before j in the path.

Proof: By induction on the length of an execution. Fix any object $o \in \mathcal{O}$.

Base case: The initializations ensure that the lemma holds initially by setting $probOwner(o)_p = home(o), \forall p \in \mathcal{P}$.

Inductive step: Assume the lemma holds for every state of any execution α of length at most l. We will show it holds for any one step extension α_1 of α. The only actions that can violate the invariant are those that modify $probOwner(o)_i$ for some $i \in \mathcal{P}$, i.e. receive$(repRead, i, o, u)_{j,i}$, request-rel-read$(o)_i$, receive$(repWrite, i, o, u, t, r)_{j,i}$, receive$(reqWrite,$ p,$o)_{j,i}$, request-rel-write$(o)_i$ and receive$(reqInv, j, o)_{j,i}$. The first type of actions sets $probOwner(o)_i = j$. From lemma 1 and part (3) of lemma 2, either (a) $probOwner(o)_j = j$, or (b) there is a $repWrite$ message in transit from j to another process p for object o and $probOwner(o)_j = p$, or (c) there is a process p such that $probOwner(o)_p = p$ and $probOwner(o)_j = p$. Therefore, the inductive hypothesis of the lemma implies that the first type of actions preserves the invariant. Actions of type request-rel-read$(o)_i$, preserve the invariant when they send a $repWrite$ message to a process p because of the inductive hypothesis and the fact that they set $probOwner(o)_i = p$. (For the same reason receive$(reqWrite, p, o)_{j,i}$ and request-rel-write$(o)_i$ also preserve the invariant). It remains to show that actions of the form request-rel-read$(o)_i$ preserve the invariant when they do not send a $repWrite$ message. We note that in this case they send a $repInv$ message to a process p and set $probOwner(o)_i = p$. Therefore, we can apply part (6) of lemma 2 and part (3) of lemma 1 to conclude that they also preserve the invariant in this case. Actions of the form receive$(repWrite, i, o, u, t, r)_{j,i}$, preserve the invariant because of the inductive hypothesis and the fact that they set $probOwner(o)_i = i$. Finally, parts (3) and (7) of lemma 1 and the inductive hypothesis imply that receive$(reqInv, j, o)_{j,i}$ also preserve the invariant.

Lemma 6: In any well-formed fair execution α of $D \times U$ for any $o \in \mathcal{O}$ and any $i, p \in \mathcal{P}$, if i receives a $(reqRead, p, o)$ (or $(reqWrite, p, o)$) message at some point in α when $probOwner(o)_i = i$ and no user is in $C(o)_w$ or $C(o)_r$, then at some later point in α a receive$(repRead, p, o, u)_{i,p}$ (or a receive$(repWrite, p, o, u, t, r)_{i,p}$) occurs or some user enters $C(o)_w$ or $C(o)_r$.

Proof: We first consider the case of a $(reqRead, p, o)$ message. If $probOwner(o)_i = i$ when a receive$(reqRead, p, o)_{j,i}$ is executed (for some $j \in \mathcal{P}$), then either (a) $held(o)_i = write$ or (b) $held(o)_i \neq write$. In case (a) i appends a $(repRead, p, o, u)$ to $out(p)_i$ and lemma 3 implies that the lemma holds. In case (b) U_i is in $T(o)_w$ or $C(o)_w$. In the latter case the lemma also holds. In the former case, we can use fairness and lemma 4 to conclude that the lemma also holds. For a $(reqWrite, p, o)$ message, if $probOwner(o)_i = i$ when a receive$(reqWrite, p, o)_{j,i}$ is executed (for some $j \in \mathcal{P}$), then either (c) $held(o)_i = false$ or (d) $held(o)_i \neq false$. Using an argument similar to the one in the previous paragraph it is easy to see that the lemma holds in case (c). In case (d), U_i is in $T(o)_w$, $C(o)_w$, $T(o)_r$ or $C(o)_r$ and once more an argument similar to the one presented before shows that the lemma also holds in this case.

Lemma 7: In any well-formed fair execution α of $D \times U$, if i sends a $(reqRead, i, o)$ (or $(reqWrite, i, o)$) message, m, at some point in α when no user is in $C(o)_w$ or $C(o)_r$, then at some later point in α the message is received by a process j such that $probOwner(o)_j = j$, or some user U_p $(p \neq i)$ enters $C(o)_w$ or $C(o)_r$.

Proof: By observing the code for receive$(reqRead, i, o)_{q,p}$ and receive$(reqWrite, i, o)_{q,p}$, we conclude that forwarding of m stops if one of these actions is executed and $held(o)_p = write$ or $probOwner(o)_p = p$. Furthermore, lemma 3 and lemma 5 imply that m eventually reaches such a process p. We consider 2 cases depending on the state of process p when forwarding of m stops, (a) $probOwner(o)_p = p$ and (b) $probOwner(o)_p \neq p$. If case (a) occurs the lemma holds, therefore consider case (b). In this case, it must be that $held(o)_p = write$, therefore p has sent a $(reqWrite, p, o)$ message to $probOwner(o)_p$. We can use the same argument as before to conclude that forwarding of this message stops when the message reaches a process q such that $held(o)_q = write$ or $probOwner(o)_q = q$. We can continue this reasoning using the same arguments as above. We argue that it is not possible for all requests to be blocked at a process with $held(o) = write$ because of lemma 5. Therefore, the lemma holds.

Theorem 3: If $D \times U$ satisfies the well-formedness condition then it satisfies the progress condition.

Proof: Fix any object $o \in \mathcal{O}$ and any fair execution α of $D \times U$. We start by proving progress for $E(o)_r$. By definition a user U_i enters $E(o)_r$ when a request-rel-read$(o)_i$ occurs. This action sets $status_i = (repRelRead, o)$ enabling reply-rel-read$(o)_i$. Since $status_i$ retains this value until reply-rel-read$(o)_i$ occurs, fairness of α implies that this action occurs, i.e. that U_i enters $R(o)$ at same later point in α. An identical argument holds for $E(o)_w$. Therefore, the theorem holds for the exit regions of any object o.

We now show progress for $T(o)_r$. By definition a user U_i enters $T(o)_r$ when a request-acq-read$(o)_i$ occurs. Observing the code for request-acq-read$(o)_i$, we conclude that if $token(o)_i \neq none$ this action sets $status_i = (repAcqRead, o)$ enabling reply-acq-read$(o)_i$. Once more fairness of α implies that reply-acq-read$(o)_i$ eventually occurs. Therefore, progress holds in this case. In the other case $(token(o)_i = none)$ a $(reqRead, i, o)$ message is appended to $out(probOwner(o))$. At this point we can apply lemmas 6 and 7 to conclude that progress also holds in this case.

To show progress for $T(o)_w$, we note that by definition a user U_i enters $T(o)_w$ when a request-acq-write$(o)_i$ occurs. If $probOwner(o)_i = i$ when this action occurs then this action sets $status_i = (repInvalidate, o)$ and appends $(reqInv, i, o)$ messages to $out(p)_i$ (for every $p \in tokenSet(o)_i$). If some user is in $C(o)_w$ or $C(o)_r$ at this point the lemma holds, therefore assume there is no such user. Lemma 4 says that i eventually receives a reply to each $reqInv$ message or some user enters $C(o)_w$

or $C(o)_r$. In the latter case the theorem holds. In the former case, receive$(repInv, i, o)_{p,i}$ will be executed repeatedly and $tokenSet(o)_i$ will eventually become empty enabling reply-invalidate$(o)_i$. Fairness implies that this action eventually occurs and sets $status_i = (repAcqWrite, o)$. Therefore, U_i eventually enters $C(o)_w$ and the lemma also holds in this case. In the other case ($probOwner(o)_i \neq i$ when the request-acq-write$(o)_i$ occurs), a $(reqWrite, i, o)$ message is appended to $out(probOwner(o))$. At this point we can apply lemmas 6 and 7 to conclude that the theorem holds.

5.4 Atomicity

In this section, we show that if U satisfies the well-formedness and synchronized data accesses conditions, then $D \times U$ also satisfies the atomicity condition.

Lemma 8: If U satisfies the well-formedness and synchronized data accesses conditions then for any execution β of $D \times U$, any $i \in \mathcal{P}$ and any $o \in \mathcal{O}$, when a reply-acq-write$(o)_i$ (or reply-acq-read$(o)_i$) action π occurs in β the following holds: $\forall c \in \mathcal{O}$: if request-write$(c, v)_j$ is the last request-write action with argument c that precedes π in β then $value(c)_i = v$ or, if there is no such action, $value(c)_i = v_o$.

Proof: By induction on the number of reply-acq-write(o) actions that precede π in an execution β.

Base case: Zero occurrences of reply-acq-write(o) actions before point π in β. The synchronized data accesses condition together with the fact that the sets in $\mathcal{O}$ are pairwise disjoint imply that no request-write$(c, v)_j$ can occur before π for any $c \in o, j \in \mathcal{P}$ and $v \in \mathcal{V}$. Therefore, we must show that if a reply-acq-write$(o)_i$ (or reply-acq-read$(o)_i$) action occurs at point π in β then, at this point, $\forall \in c :\ value(c)_i = v_o$. This holds because the initializations ensure that $\forall c \in \mathcal{C}, p \in \mathcal{P}$: $value(c)_p = v_o$.

Inductive step: Assume the lemma holds up to point π_o where the *l-th* reply-acq-write(o) occurs in β. We will show it also holds at point π_2 just after the the *l+1-th* reply-acq-write(o) (if it exists) and just after any reply-acq-read(o) between π_o and π_2.

Assume the *l-th* reply-acq-write(o) occurs at port j leading U_j into $C(o)_w$. The exclusion condition says that no reply-acq-write$(o)_i$ (or reply-acq-read$(o)_i$) can occur at any port i until a request-rel-write$(o)_j$ occurs. If this request-rel-write$(o)_j$ never occurs in β then the lemma holds. Therefore, assume it occurs at point π_1 in β.

We note that for any cell $c \in o$, $value(c)_j$ can only change if an action of one of the following forms occurs: request-write$(c, v)_j$, receive$(repWrite, j, o, u, t, r)_{p,j}$ and receive$(repRead, j, o, u)_{p,j}$. We also note that well-formedness prevents $status_j$ from being equal to $(reqAcqRead, o)$ or $(reqAcWrite, o)$ while U_j is in $C(o)_w$. Therefore, actions of the last two types can not modify the values of cells in o between points π_o and π_1. The synchronized data accesses and exclusion condition together with the fact that elements of $\mathcal{O}$ are pairwise disjoint, imply that no request-write$(c, v)_p$, for $p \neq j$ can occur between π_o and π_1. These claims together with the inductive hypothesis imply that $\forall c \in \mathcal{O}$ if request-write$(c, v)_j$ is the last request-write action with argument c that precedes π_1 in β then $value(c)_j = v$ or, if there is no such operation, $value(c)_j = v_o$.

The synchronized data accesses condition and the fact that elements of $\mathcal{O}$ are pairwise disjoint prevents actions of the form request-write$(c, v)_p$ ($\forall c \in o, p \in \mathcal{P}$) from occurring between π_1 and π_2. Therefore, to show the lemma holds for the *l+1-th* reply-acq-write$(o)_i$ and all reply-acq-read$(o)_i$ actions between π_o and π_2, we must show that when any of these actions occurs for any cell $c \in o$ $value(c)_i$ is equal to $value(c)_j$ at point π_1.

Claim 1: The proof of theorem 1 shows that if U_j is in $C(o)_w$ then $token(o)_j = write$ and $held(o)_j = write$. From lemma 1 this implies that if U_j is in $C(o)_w$ then $probOwner(o)_j = j$. Observing the code for request-rel-write$(o)_j$, we conclude that $probOwner(o)_j = j$ at point π_1. We argue that $probOwner(o)_j$ retains its value up to the point π'_1 where j sends a $repWrite$ message for object o to another process. The only actions that can violate this are prevented from doing so by the fact that $probOwner(o)_j = j$ (as shown by lemmas 1 and 2). Furthermore, when j sends a $repWrite$ message to another process p, it sets $token(o)_j = none$ and $probOwner(o)_j = p$. From lemma 1 it must acquire a write token for o from another process, before a subsequent reply-acq-write$(o)_j$ occurs. Lemma 1 implies that p has $held(o)_p = write$ at point π'_1 and will keep $held(o)_p = write$ until after a reply-acq-write$(o)_p$ occurs. Therefore, after point π'_1 no token transfer requests will be serviced until after a reply-acq-write$(o)_p$ occurs.

Consider two cases (a) the *l+1-th* reply-acq-write(o) operation occurs at port j and (b) it does not. Claim 1 implies that in case (a) π'_1 must occur after π_2. We argue that $value(c)_j$ $(\forall c \in o)$ is not modified between π_1 and π_2 in this case. This is so because, as discussed above, no request-write$(c, v)_j$ $(\forall c \in o)$ can occur, and the fact that $probOwner(o)_j = j$ and lemmas 1 and 2 imply that the arrival of $repRead$ and $repWrite$ messages can not change the values of the cells in o. Therefore, the lemma holds in this case for the *l+1-th* reply-acq-write(o) operation. We will also show it holds for all the reply-acq-read$(o)_i$ operations between π_o and π_2. We consider two cases (a.1) $i = j$ and (a.2) $i \neq j$. The lemma holds in the first case as discussed earlier. In case (a.2), a reply-acq-read$(o)_i$ can only occur (from lemma 2) if $token(o)_i = read$ and since $token(o)_i = none$ at point π_1 (from lemma 1 and the fact that $token(o)_j = write$), i must receive a $repRead$ message from j. This message must be sent before π_2 and after π_1, therefore it will carry the state of the cells of o at j at point π_1. The receive action for this $repRead$ message ensures that the lemma holds by setting the value of each cell of o in i equal to the value received in the message.

In case (b), the *l+1-th* reply-acq-write$(o)_i$ can only occur if $token(o)_i = write$ (lemma 1) and since $token(o)_i = none$ at point π_1 (from lemma 1 and the fact that $token(o)_j = write$), i must receive a $repWrite$ message from j. This message is sent at point π'_1. We can use an argument identical to that in case (a) to conclude that $value(c)_j$ $(\forall c \in o)$ is not modified between π_1 and π'_1. Therefore, the message will carry the state of the cells of o at j at point π_1 and like in the read case the receive action ensures that the lemma holds. To show it also holds for all the reply-acq-read$(o)_i$ operations between π_o and π_2, we note that any $repRead$ message enabling such an action must be sent between π_1 and π'_1. Therefore, we can use the same argument we used in cases (a.1) and (a.2) to complete the proof of the lemma.

Theorem 3: If U satisfies the well-formedness and the synchronized data accesses conditions then $D \times U$ satisfies the atomicity condition.

Proof: We pick as serialization point for each (request-read$(c)_i$, reply-read$(c, v)_i$) operation the point where the reply-read$(c, v)_i$ action occurs in γ. Similarly, for each (request-write$(c, v)_i$, reply-write$(c)_i$) operation, we pick as serialization point the point where the request-write$(c, v)_i$ action occurs in γ. It is clear that the serialization point for each operation is between its request and reply actions, as required. For incomplete operations we pick the point where the request action occurs as the serialization point and for read operations we pick as response $value(c)_i$ at that point.

We must now show that the sequence γ', obtained from γ by moving the request and reply actions of each operation op to $*_{op}$ in this order, is a trace of $\mathcal{T}$. From the definition of $\mathcal{T}$, it is enough to show that each reply-read(c, v) in γ' returns the value v written by the last request-write(c, v) that precedes the read in γ' (or $v = v_o$ if there is no such action).

We note that with the serialization points chosen the relative order between the reply-read(c, v) actions and the request-write(c, v) actions is identical in γ' and β. Therefore, we can use lemma 8 to prove the theorem. Lemma 8, the well-formedness, synchronized data accesses and exclusion conditions together with the fact that the sets in $\mathcal{O}$ are pairwise disjoint imply the theorem.

6 Complexity analysis

In this section we analyse the number of messages sent and the time elapsed between the occurrence of a request-acq-write$(o)_i$ (or request-acq-read$(o)_i$) and the corresponding reply-acq-write$(o)_i$ (or reply-acq-read$(o)_i$). We consider only the case of an isolated request, i.e. we assume that between the point where the request occurs and the point where the reply occurs $status_p = idle$ (for all $p \in \mathcal{P} - \{i\}$), and that no user is in $C(o)_w$ or $C(o)_r$.

The maximum number of messages sent between the occurrence of a request-acq-read$(o)_i$ and the corresponding reply under these conditions is n. This case occurs if the path defined by the $probOwner(o)$ variables starting at i has maximum length. In this case, it takes $n-1$ messages for the request to reach the process j such that $probOwner(o)_j = j$ plus the reply sent by j to i. For a request-acq-write$(o)_i$, the worst case message complexity occurs if the path defined by the $probOwner(o)$ variables that connects i to j (such that $probOwner(o)_j = j$) has length 2 and there are $n-2$ processes different from i and j that have $token(o)$ variables equal to $read$. In this case, it takes 3 messages for i to receive its reply, and $2(n-2)$ messages to invalidate the $n-2$ read tokens. Note that processes that have $token(o) = read$ have their $probOwner(o)$ variables pointing directly to the owner, therefore this is really the worst case.

For the time complexity analysis, we assume that l is an upper bound on time for each process task and d is an upper bound on the time to deliver the oldest message in a channel. The maximum time between the occurrence of a request-acq-read$(o)_i$ and the corresponding reply occurs in the same scenario as the worst case message complexity for this type of request. Therefore, the total elapsed time is $n(d+l)$. For a request-acq-write$(o)_i$, the worst case time complexity does not occur for the same scenario as the worst case message complexity because invalidations proceed in parallel. In the worst case scenario, the path defined by the $probOwner(o)$ variables has maximal length and the next to last process in the path has $token(o) = read$. The time complexity in this case is $(n+2)(d+l)$. $n(d+l)$ time for a $repWrite$ message to arrive at i and $2(d+l)$ to invalidate the $read$ token.

References

[1] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 93 COMPCON Conference*, pages 528–537, February 1993.

[2] Miguel Castro. Distributed Shared Object Memory. Master's thesis, Universidade Técnica de Lisboa, Instituto Superior Técnico, Lisboa, 1995.

[3] Paulo Guedes and Miguel Castro. Distributed Shared Object Memory. In *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, Napa, CA., October 14-15 1993. IEEE Computer Society Press.

[4] Nuno Neves, Miguel Castro, and Paulo Guedes. A Checkpoint Protocol for an Entry Consistent Shared Memory System. In *Thirteenth ACM Symposium on Principles of Distributed Computing (PODC)*, August 1994.