Specifying an Implementation to Satisfy Interface Specifications: A State Transition Approach*

Simon S. Lam** and A. Udaya Shankar***

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-12 April 1988

Abstract

We present a solution to the problem posed by Leslie Lamport to participants of the Specification Logics session in the 1987 Lake Arrowhead workshop. Formal specifications are given for a database interface offering serializable access to concurrent client programs, a two-phase locking implementation of the client interface, and the physical-database interface accessed by the implementation. We sketch a proof that the implementation satisfies the client interface specification, assuming that the physical-database interface specification holds.

*Presented at the 26th Annual Lake Arrowhead workshop, Sept. 16-18, 1987.

**Work supported by National Science Foundation under grant no. NCR-8613338.

***Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. Work supported by National Science Foundation under grant no. ECS-8502113.
# TABLE OF CONTENTS

1 INTRODUCTION ........................................................................................................... 1
2 UPPER INTERFACE SPECIFICATION ......................................................................... 3
   2.1 State variables ................................................................................................. 3
   2.2 State functions ............................................................................................... 3
   2.3 Events ............................................................................................................... 4
   2.4 Safety requirements ......................................................................................... 6
   2.5 Progress requirements ................................................................................. 6
3 LOWER INTERFACE SPECIFICATION .................................................................... 7
   3.1 State variables ................................................................................................. 7
   3.2 State functions ............................................................................................... 7
   3.3 Events ............................................................................................................... 7
   3.4 Safety requirements ......................................................................................... 8
   3.5 Progress requirements ..................................................................................... 8
4 TWO-PHASE LOCKING IMPLEMENTATION ............................................................... 9
   4.1 State variables ................................................................................................. 9
   4.2 State functions ............................................................................................... 9
   4.3 Events ............................................................................................................... 9
5 VERIFICATION ........................................................................................................ 11
   5.1 Proof of refinement ......................................................................................... 12
   5.2 Proof of serializability .................................................................................... 13
   5.3 Proof of progress ............................................................................................. 13
   5.4 Conclusion ...................................................................................................... 17
REFERENCES ............................................................................................................. 17
1. INTRODUCTION

Consider the database system illustrated in Figure 1. Each client program performs a sequence of transactions. Client programs can execute concurrently. We refer to the interface between the client programs and the two-phase locking system as the upper interface. We refer to the interface between the two-phase locking system and the physical database as the lower interface.

![Diagram of the database system]

Figure 1. A database system.

**Interface specification**

The informal specification of each interface consists of a set of procedures that can be executed concurrently. We will specify each interface by an event-driven system together with some safety and progress requirements. An event-driven system consists of a set of state variables and a set of events. Each event is specified by an enabling condition and an atomic action. The enabling condition is a predicate in the state variables. The action specifies updates to the state variables when the event occurs.

An interface procedure $P$ is modeled as two events: $Call(P)$ and $Return(P)$. Since several invocations to $P$ can be concurrently active, it is necessary to tag each call of $P$ with a unique identifier, which will be used in the corresponding return of $P$. Therefore each interface procedure $P$ is modeled by the two events: $Call(i,P)$ and $Return(i,P)$, where the identifier $i$ must be unique for all possible concurrent invocations of $P$.

In summary, an interface specification includes the following:

(a) a set of state variables (including history variables) and state functions;
(b) a $Call(i,P)$ event and a $Return(i,P)$ event for each interface procedure $P$;
(c) a set of safety requirements;
(d) a set of progress requirements.

We note that state functions can always be transformed into state variables. Also, the event-driven system in the specification can be very small. In the extreme case, a single state variable is enough, i.e., a history variable recording the sequence of all procedure calls and returns; each event is always enabled and its action consists of only updating the history variable. For the interfaces to be specified in this paper, we found that some safety requirements can be more easily expressed by state variables and the updating of state variables than by formulating assertions on a history variable.

**Implementation specification**

An implementation of the two-phase locking system is specified by an event-driven system. It is a "refinement" of the upper interface specification, obtained as follows [5]:

(a) Additional state variables are introduced, augmenting those in the upper interface specification. Thus, there is a projection mapping from each state of the implementation to a state of the upper interface, refered to as its image at the interface [2,3]. Some of the
variables of the upper interface specification can be declared to be auxiliary (e.g., a history variable).

(b) The upper interface events are refined and additional events are defined. The events can include in their enabling conditions and actions, the state variables introduced in part (a), as well as call and return events of the lower interface.

Each implementation event $e_I$ must be a refinement of some upper interface events, which means that if $e_I$ can take the implementation from state $s_1$ to $s_2$, then there is an upper interface event $e_U$ that can take the upper interface from state $t_1$ to $t_2$, where $t_1$ is the image of $s_1$. This condition can be relaxed by introducing a safety requirement $S$, in which case the condition has to be satisfied only for each $(s_1, s_2)$ pair such that $s_1$ and $s_2$ satisfy $S$. We will have to prove that such safety requirements introduced are in fact safety properties of the implementation. A special case of event refinement is that $e_I$ has a null image (i.e., $t_1$ equals $t_2$).

The implementation is a refinement of the upper interface if all implementation events are refinements of upper interface events. In this case, safety properties of the event-driven system of the upper interface are also safety properties of the implementation [2,5].

Specifying events by predicates

Consider a system with state variables $\{v_i\}$. The enabling condition of an event is specified by a predicate in $\{v_i\}$. Instead of specifying the event's action by algorithmic code, we use a predicate in $\{v_i\} \cup \{v'_i\}$, where $v_i$ denotes the value of a state variable immediately before the event occurrence, and $v'_i$ denotes its value immediately after the event occurrence [4]. For brevity, if $v'_i$ does not appear in an event's definition, then $v'_i = v_i$ is implicitly assumed. For example, an event $e_I$ that is enabled whenever the state variable $v_2$ is less than 5 and whose action increments the state variable $v_1$ by 1 is defined by $e_I \Leftrightarrow (v_2 < 5 \land v'_1 = v_1 + 1)$.

Checking implementation events

Specifying events by predicates makes it easy to check if implementation events are refinements of upper interface events [5]. Event $e_I$ is a refinement of the upper interface events, $e_1, e_2, \ldots, e_n$, if $e_I \Rightarrow e_1 \lor e_2 \lor \cdots \lor e_n$. Given the safety requirement $S$, $e_I$ is a refinement if $S \land e_I \Rightarrow e_1 \lor e_2 \lor \cdots \lor e_n$.

For most implementation events, $e_I$ is a refinement of a single upper interface event $e_U$. In this case, we need only check either $e_I \Rightarrow e_U$ or $S \land e_I \Rightarrow e_U$.

Verification of an implementation

Having an implementation that is a refinement of the upper interface, it remains to show that the implementation satisfies the following:

(i) Safety requirements that are not safety properties of the event-driven system of the upper interface, e.g., serializability.

(ii) Progress requirements in the upper interface specification.

For the two-phase locking implementation, we found that it is actually easier to give a direct proof that the implementation satisfies the progress requirements in the upper interface specification than to give a proof via the projection mapping. Our progress proof employs a novel metric based upon lexicographic ordering.
2. UPPER INTERFACE SPECIFICATION

Define the following constants. Let OBJECTS denote the set of objects in the database, VALUES the set of values each object can have, KEYS the set of keys, and IDS the set of transaction identifiers. The entries of IDS are needed to specify correct usage of keys. They are also adequate as identifiers in interface procedure calls, since each transaction has at most one procedure call outstanding. For each \( obj \in \text{OBJECTS} \), let its initial value be given by \( \text{INITVALUE}(obj) \). We will use \( key, \ obj, \ val, \ id \) as variables that range over the corresponding sets.

We say that a transaction has a procedure invocation \textit{outstanding} if it has called the procedure and not yet returned. We say that the transaction is \textit{active} if it has returned from a \( \text{BeginTr} \) call with a key, and it has not yet ended.

2.1. State variables

\( H_U : \) sequence of \( \{(id, \text{BeginTr}, key, obj, val), (id, \text{ReadTr}, key, obj, val, \text{OK}), (id, \text{WriteTr}, key, obj, val), (id, \text{EndTr}, key, \text{OK}), (id, \text{AbortTr}, key)\} \).

Initially, \( H_U \) is the null sequence.

History of the returns of procedure invocations. The \( (id, \text{AbortTr}, key) \) entry is used to record all returns aborting transactions. The other entries indicate successful returns. An unsuccessful \( \text{BeginTr} \) return is not recorded in \( H_U \). \( H_U \) is adequate for stating serializability.

\( \text{status}_{U}(id) : \{\text{NOTBEGUN, READY, COMMITTED, ABORTED}\} \cup \{(\text{BeginTr}), (\text{ReadTr}, key, obj), (\text{WriteTr}, key, obj, val), (\text{EndTr}, key), (\text{AbortTr}, key)\} \).

Initially, \( \text{status}_{U}(id) = \text{NOTBEGUN} \).

Indicates the status of transaction \( id \). \( \text{NOTBEGUN} \) means that the transaction has not yet issued a \( \text{BeginTr} \) call, or such a call returned with \( \text{FAILED} \). \( \text{READY} \) means that the transaction is active and has no interface procedure invocation outstanding. A procedure call, such as \( (\text{ReadTr}, key, obj) \), means that the transaction is active and has that procedure invocation outstanding. \( \text{COMMITTED} \) means that the transaction has ended successfully. \( \text{ABORTED} \) means that the transaction has ended by aborting.

\text{allocated}(key) : \) boolean. Initially false.

True iff \( key \) is allocated to a transaction.

When we refer to a tuple in the domain of \( \text{status}_{U}(id) \), such as \( (\text{ReadTr}, key, obj) \), where a component in the tuple can have any of its allowed values, we shall omit that component in our reference. For example, \( \text{status}_{U}(id) = (\text{ReadTr}, obj) \) means \( \text{status}_{U}(id) = (\text{ReadTr}, key, obj) \) for some value of \( key \). More than one component in a tuple may be omitted. For example, \( (obj) \) refers to \( (\text{ReadTr}, key, obj) \) for some \( key \) or \( (\text{WriteTr}, key, obj, val) \) for some \( key \) and some \( val \). The same notational abbreviation will be used in referring to elements of \( H_U \). For example, \( (id, obj) \in H_U \) means that \( H_U \) has a \( (id, \text{ReadTr}, obj, key, val) \) or a \( (id, \text{WriteTr}, obj, key, val, \text{OK}) \) entry for some \( key \) and some \( val \).

2.2. State functions

\text{active}(id) : \) boolean

True iff \( (id, \text{BeginTr}) \in H_U \), and neither \( (id, \text{EndTr}) \) nor \( (id, \text{AbortTr}) \) is in \( H_U \).

\text{accessed}(id) : \) powerset of OBJECTS

The set of objects that have been accessed by an active transaction \( id \).

\( = \emptyset \), if \( \neg \text{active}(id) \).
\[
\{ \text{obj : } status_U(id) = \text{obj} \land (id, \text{obj}) \in H_U \}, \text{ if active (id).}
\]

\text{concurrentaccess (id) : boolean}
\quad \text{True iff there is an } i \in IDS - \{id\} \text{ such that } accessed(i) \cap accessed(id) \text{ is not empty.}

\text{committedvalue (obj) : VALUES}
\quad = \text{INITVALUE(obj), if there is no (id, WriteTr, obj) } \in H_U \text{ such that } status_U(id) = \text{COMMITTED.}
\quad = \text{val, if there is an id such that } status_U(id) = \text{COMMITTED and } H_U \text{ contains a (id, WriteTr, obj) entry, and (id, WriteTr, obj, val) is the last such entry.}

\text{currentvalue (obj, id) : VALUES } \cup \{ \text{NULL} \}
\quad = \text{NULL, if } \neg \text{active (id).}
\quad = \text{committedvalue (obj), if active (id) and (id, WriteTr, obj) } \notin H_U.
\quad = \text{val, if active (id), there is a (id, WriteTr, obj) entry in } H_U, \text{ and (id, WriteTr, obj, val) is the last such entry.}

2.3. Events

For readability, we model each procedure return by two return events, one for success and one for abort. Also, the enabling condition of an event is placed on the first line of the definition.

\text{Call (id, BeginTr) } \equiv
\quad \text{status}_U(id) = \text{NOTBEGUN}
\quad \land \text{status}_U(id)' = \text{(BeginTr)}

\text{Return (id, BeginTr, key) } \equiv
\quad \text{status}_U(id) = \text{(BeginTr)} \land \neg \text{allocated (key)}
\quad \land \text{status}_U(id)' = \text{READY}
\quad \land \text{allocated (key)'}
\quad \land H_U' = H_U \circ (id, \text{BeginTr, key})

\text{Return (id, BeginTr, FAILED) } \equiv
\quad \text{status}_U(id) = \text{(BeginTr)} \land (\forall \text{key : allocated (key)})
\quad \land \text{status}_U(id)' = \text{NOTBEGUN}

\text{Call (id, ReadTr, key, obj) } \equiv
\quad \text{status}_U(id) = \text{READY} \land \text{allocated (key)}
\quad \land \text{status}_U(id)' = \text{(ReadTr, key, obj)}

\text{Return (id, ReadTr, key, obj, val) } \equiv
\quad \text{status}_U(id) = \text{(ReadTr, key, obj)}
\quad \land \text{status}_U(id)' = \text{READY}
\quad \land \text{val = currentvalue (obj, id)}
\quad \land H_U' = H_U \circ (id, \text{ReadTr, key, obj, val})
Return \((id, \text{ReadTr}, key, obj, \text{ABORT})\) \(\equiv\)
\[\text{status}_U(id) = (\text{ReadTr}, key, obj) \land \text{concurrentaccess}(id)\]
\[\land \text{status}_U(id') = \text{ABORTED}\]
\[\land \neg \text{allocated}(key')\]
\[\land H_U' = H_U @ (id, \text{AbortTr}, key)\]

Call \((id, \text{WriteTr}, key, obj, val)\) \(\equiv\)
\[\text{status}_U(id) = \text{READY} \land \text{allocated}(key)\]
\[\land \text{status}_U(id') = (\text{WriteTr}, key, obj, val)\]

Return \((id, \text{WriteTr}, key, obj, val, \text{OK})\) \(\equiv\)
\[\text{status}_U(id) = (\text{WriteTr}, key, obj, val)\]
\[\land \text{status}_U(id') = \text{READY}\]
\[\land H_U' = H_U @ (id, \text{WriteTr}, key, obj, val, \text{OK})\]

Return \((id, \text{WriteTr}, key, obj, val, \text{ABORT})\) \(\equiv\)
\[\text{status}_U(id) = (\text{WriteTr}, key, obj, val) \land \text{concurrentaccess}(id)\]
\[\land \text{status}_U(id') = \text{ABORTED}\]
\[\land \neg \text{allocated}(key')\]
\[\land H_U' = H_U @ (id, \text{AbortTr}, key)\]

Call \((id, \text{EndTr}, key)\) \(\equiv\)
\[\text{status}_U(id) = \text{READY} \land \text{allocated}(key)\]
\[\land \text{status}_U(id') = (\text{EndTr}, key)\]

Return \((id, \text{EndTr}, key, \text{OK})\) \(\equiv\)
\[\text{status}_U(id) = (\text{EndTr}, key)\]
\[\land \text{status}_U(id') = \text{COMMITTED}\]
\[\land \neg \text{allocated}(key')\]
\[\land H_U' = H_U @ (id, \text{EndTr}, key, \text{OK})\]

Return \((id, \text{EndTr}, key, \text{ABORT})\) \(\equiv\)
\[\text{status}_U(id) = (\text{EndTr}, key) \land \text{concurrentaccess}(id)\]
\[\land \text{status}_U(id') = \text{ABORTED}\]
\[\land \neg \text{allocated}(key')\]
\[\land H_U' = H_U @ (id, \text{AbortTr}, key)\]

Call \((id, \text{AbortTr}, key)\) \(\equiv\)
\[\text{status}_U(id) = \text{READY} \land \text{allocated}(key)\]
\[\land \text{status}_U(id') = (\text{AbortTr}, key)\]

Return \((id, \text{AbortTr}, key)\) \(\equiv\)
\[\text{status}_U(id) = (\text{AbortTr}, key)\]
\[\land \text{status}_U(id') = \text{ABORTED}\]
\[\land \neg \text{allocated}(key')\]
\[\land H_U' = H_U @ (id, \text{AbortTr}, key)\]
2.4. Safety requirements

The interface system events ensure that each transaction issues a correct sequence of procedure calls. Formally, define the following state function:

\( \text{legal}(id): \text{boolean} \)

True iff the subsequence of \((id)\) entries in \(H_U\) is a prefix of \((id, \text{BeginTr})\@<\text{successes}>\@<\text{final}>\), where \(<\text{successes}>\) is a sequence of zero or more \((id, \text{obj})\) entries, and \(<\text{final}>\) is either \((id, \text{AbortTr})\) or \((id, \text{EndTr})\).

It can be proved that \(\text{legal}(id)\) is a safety property of the event-driven system in the interface specification. (Proof omitted.)

An invocation of \(\text{ReadTr}, \text{WriteTr}, \text{or EndTr}\) by transaction \(id\) aborts only if it accesses an object that is also accessed by another concurrently executing transaction. This has been captured formally by including \(\text{concurrentaccess}(id)\) in the enabling conditions of the corresponding return events.

The definition of \(\text{committedvalue}\) ensures that writes of aborted transactions do not influence the committed values. The definition of \(\text{currentvalue}\) ensures that the values read by a transaction are not affected by writes of other concurrently executing transactions. Observe that \(\text{currentvalue}(\text{obj}, id_1)\) can differ from \(\text{currentvalue}(\text{obj}, id_2)\), for two concurrently executing transactions \(id_1\) and \(id_2\).

Let us review some basic definitions from serializability theory [1]. The committed history \(C(H_U)\) is the subsequence of \(H_U\) obtained by including all \((id)\) entries such that \(\text{status}_U(id) = \text{COMMITTED}\).

For any two transactions \(id_1\) and \(id_2\), define the boolean function \(\text{dependency}(id_1, id_2)\) to be true iff for some \(\text{obj}, (id_1, \text{WriteTr}, \text{obj})\@(id_2, \text{obj})\) or \((id_1, \text{obj})\@(id_2, \text{WriteTr}, \text{obj})\) is a subsequence of \(C(H_U)\).

We say that \(\text{dependency}\) is acyclic if for every \(n \geq 2\), there does not exist distinct \(id_1, id_2, \ldots, id_n\), such that \(\text{dependency}(id_k, id_{k+1})\), for \(k = 1, \ldots, n-1\), and \(\text{dependency}(id_n, id_1)\). A fundamental theorem of serializability is that \(H_U\) is serializable iff \(\text{dependency}\) is acyclic [1].

Define the following state functions:

\(\text{keyof}(id): \text{KEYS} \cup \\{\text{NULL}\}\)

\(=\text{NULL}, \text{if} \neg \text{active}(id).\)

\(=\text{key}, \text{if} \text{active}(id) \text{and} (id, \text{BeginTr}, \text{key}) \text{in} H_U.\)

\(\text{correctkeyuse: boolean.}\)

True iff every transaction has used the correct key in all its procedure calls, i.e., every \((id, \text{key}) \in H_U\) satisfies \(\text{key} = \text{keyof}(id)\).

The upper interface specification includes the following:

\(\text{Safety requirement: correctkeyuse } \Rightarrow \text{dependency} \text{ is acyclic.}\)

2.5. Progress requirements

The progress guarantee that every procedure call eventually returns is formally specified by:

\(L_1 \equiv \text{status}_U(id) \in \{(\text{BeginTr}), (\text{ReadTr}), (\text{WriteTr}), (\text{EndTr}), (\text{AbortTr})\}\)

leads to \(\text{status}_U(id) \in \{\text{READY}, \text{ABORTED}, \text{COMMITTED}, \text{NOTBEGUN}\}\)

The assumption that every active transaction that does not abort eventually issues an \(\text{EndTr}\) call can be stated as follows: If every \(\text{ReadTr}\) and \(\text{WriteTr}\) call made by the transaction
returns successfully, then the transaction will eventually issue an EndTr call. Formally:

\[ L_2 \equiv (status_U(id) \in \{(ReadTr),(WriteTr)\}) \text{ leads } \text{to} \ status_U(id) = \text{READY} \]

\[ \Rightarrow (status_U(id) = \text{READY} \text{ leads } \text{to} \ status_U(id) = \text{EndTr}) \]

The upper interface specification includes the following:

**Progress requirement:** correctkeyuse \& L_2 \Rightarrow L_1.

3. **LOWER INTERFACE SPECIFICATION**

Note that outstanding procedure calls at the lower interface can be uniquely identified by the entries of KEYS.

3.1. **State variables**

\[ status_L(key) = \{\text{READY, (AcqLock, obj), (RelLock, obj), (Read, obj), (Write, obj, val)}\} \]

Initially READY.

Indicates the status of any procedure invocation identified by key. READY means that key has no lower interface procedure invocation outstanding. Otherwise, indicates the outstanding procedure invocation.

\[ owned(key, obj) : \text{boolean. Initially false.} \]

True iff key has locked obj.

\[ storedvalue(obj) : \text{VALUES. Initially, storedvalue(obj) = INITVALUE(obj).} \]

The value of the object in the physical database.

3.2. **State functions**

\[ waiting(key, obj) : \text{boolean.} \]

True iff \[ status_L(key) = \text{(AcqLock, obj)}. \] Defined for notational convenience.

\[ deadlock(key, obj) : \text{boolean.} \]

True iff there is a cycle including the edge \( (key, obj) \) in the directed graph of nodes KEYS \( \cup \) OBJECTS, and edges \( \{(x,k) : owned(k,x)\} \cup \{(k,x) : waiting(k,x)\} \).

3.3. **Events**

The events of the interface are the calls and returns of the interface procedures AcqLock, RelLock, Read, and Write.

\[ \text{Call(key,AcqLock,obj) } \equiv \]

\[ status_L(key) = \text{READY} \]

\[ \land status_L(key)’ = \text{(AcqLock, obj)} \]

\[ \text{Return(key,AcqLock,obj,GRANTED) } \equiv \]

\[ status_L(key) = \text{(AcqLock, obj)} \land (\forall k : owned(k, obj)) \]

\[ \land status_L(key)’ = \text{READY} \]

\[ \land owned(key, obj)’ \]
Return (key, AcqLock, obj, REJECTED) ≡
status₇(key) = (AcqLock, obj) ∧ deadlock(key, obj)
∧ status₇(key)’ = READY

Call (key, RelLock, obj) ≡
status₇(key) = READY
∧ status₇(key)’ = (RelLock, obj)

Return (key, RelLock, obj) ≡
status₇(key) = (RelLock, obj) ∧ owned(key, obj)
∧ status₇(key)’ = READY
∧ ¬owned(key, obj)’

Call (key, Read, obj) ≡
status₇(key) = READY
∧ status₇(key)’ = (Read, obj)

Return (key, Read, obj, val) ≡
status₇(key) = (Read, obj)
∧ status₇(key)’ = READY
∧ val = storedValue(obj)

Call (key, Write, obj, val) ≡
status₇(key) = READY
∧ status₇(key)’ = (Write, obj, val)

Return (key, Write, obj, val) ≡
status₇(key) = (Write, obj, val)
∧ status₇(key)’ = READY
∧ storedValue(obj)’ = val

3.4. Safety requirements

The enabling condition of Return(key, AcqLock, obj, GRANTED) ensures that obj is not owned by any other key. Its action updates owned(key, obj) to true. The enabling condition of Return(key, RelLock, obj) ensures that obj is owned by key. Its action updates owned(key, obj) to false. No other event updates owned(key, obj).

The enabling condition of Return(key, AcqLock, obj, REJECTED) ensures that (key, obj) is involved in a deadlock.

3.5. Progress requirements

The lower layer guarantees the progress properties Q₁ through Q₄:

Q₁ ≡ status₇(key) = (Read) leads-to status₇(key) = READY
Q₂ ≡ status₇(key) = (Write) leads-to status₇(key) = READY
Q₃ ≡ status₇(key) = (RelLock, obj) ∧ owned(key, obj)
leads-to status₇(key) = READY ∧ ¬owned(key, obj)
\[ Q_4 \iff R_4 \implies G_4 \text{ where} \]
\[ R_4 \equiv \text{waiting}(k_1, \text{obj}) \land \text{owned}(k_2, \text{obj}) \text{ leads } \to \neg \text{owned}(k_2, \text{obj}) \]
\[ G_4 \equiv \text{waiting}(k_1, \text{obj}) \text{ leads } \to \neg \text{waiting}(k_1, \text{obj}) \]

\( Q_4 \) specifies the property that every call to \textit{AcqLock} will eventually return provided that every granted lock is eventually returned.

4. TWO-PHASE LOCKING IMPLEMENTATION

The two-phase locking implementation is obtained from the upper interface system by adding state variables, refining the upper interface events, and adding new events. The events can include events of the lower interface.

4.1. State variables

In addition to the upper interface state variables \( H_U \), \( status_U \), and \( allocated \), we add the following:

\( locked(\text{key}, \text{obj}) \): boolean. Initially false.

Indicates whether \text{key} has locked \text{obj}.

\( localvalue(\text{obj}, \text{key}) \): VALUES \cup \{\text{NULL}\}. Initially NULL.

The current value of \text{obj} as seen by transaction using \text{key}.

The upper interface variable \( H_U \) becomes auxiliary. This also makes auxiliary all state functions defined in terms of \( H_U \), such as \textit{concurrentaccess}, \textit{currentvalue}, \textit{committedvalue}, etc. An event cannot use an auxiliary variable or function in its enabling condition or in its update of a nonauxiliary variable.

4.2. State functions

\( holdinglocks(\text{key}) \): boolean.

True iff \( locked(\text{key}, \text{obj}) \) is true for some \text{obj}.

4.3. Events

Implementation events that are refinements of the upper interface events are listed first. In an event predicate, the notation \(<\text{previous definition}>\) refers to the event’s predicate definition given in Section 2.3. This notation is used whenever the refinement consists of adding conjuncts only. When the refinement is not of this simple form, we add a safety requirement which will have to be proved later.

\[ \text{Call}(id, \text{BeginTr}) \equiv <\text{previous definition}> \]

\[ \text{Return}(id, \text{BeginTr}, result) \equiv \]
\[ status_U(id) = (\text{BeginTr}) \]
\[ \land (\exists \text{key} \colon \neg \text{allocated}(\text{key}) \land \neg \text{holdinglocks}(\text{key}) \]
\[ \land \text{allocated}(\text{key})' \land status_U(id)' = \text{READY} \land result = \text{key} \]
\[ \land H_U' = H_U @ (id, \text{BeginTr}, \text{key}) \]
\[ \lor (\forall \text{key} \colon \text{allocated}(\text{key}) \]
\[ \land status_U(id)' = \text{NOTBEGUN} \land result = \text{FAILED} \land H_U' = H_U) \]

The above event is a refinement of the upper interface events, \text{Return}(id, \text{BeginTr}, \text{key}) and \text{Return}(id, \text{BeginTr}, \text{FAILED}). We have combined the two returns in the implementation because
it facilitates the progress proof; specifically, \( \text{status}_U(id) = \text{BeginTr} \) ensures that \( \text{Return}(id, \text{BeginTr}) \) is continuously enabled, and therefore will eventually occur.

\[
\text{Call}(id, \text{ReadTr}, key, obj) \equiv \text{<previous definition>}
\]

\[
\text{Return}(id, \text{ReadTr}, key, obj, val) \equiv \\
\text{status}_U(id) = \text{(ReadTr}, key, obj) \land \text{localvalue}(obj, key) \neq \text{NULL} \\
\land \text{status}_U(id)' = \text{READY} \\
\land H_U' = H_U @ (id, \text{ReadTr}, key, obj, val) \\
\land \text{val} = \text{localvalue}(obj, key)
\]

In order for the above to be a refinement, we specify the following safety requirement:

\[
A_1 \equiv \text{keyof}(id) = key \land \text{localvalue}(obj, key) \neq \text{NULL} \\
\implies \text{localvalue}(obj, key) = \text{currentvalue}(obj, id)
\]

\[
\text{Return}(id, \text{ReadTr}, key, obj, \text{ABORT}) \equiv \\
\text{status}_U(id) = \text{(ReadTr}, key, obj) \land \text{Return}(key, \text{AcqLock}, obj, \text{REJECTED}) \\
\land \text{status}_U(id)' = \text{ABORTED} \\
\land H_U' = H_U @ (id, \text{AbortTr}, key) \\
\land \neg \text{allocated}(key)' \\
\land (\forall x: \text{localvalue}(x, key)' = \text{NULL})
\]

For the above to be a refinement, it is sufficient if \( \text{concurrentaccess}(id) \) is true whenever \( \text{Return}(key, \text{AcqLock}, obj, \text{REJECTED}) \) occurs. Because the latter event is enabled only when \( \text{deadlock}(key, obj) \) is true, the following safety requirement is adequate:

\[
A_2 \equiv \text{keyof}(id) = key \land \text{deadlock}(key, obj) \implies \text{concurrentaccess}(id)
\]

\[
\text{Call}(id, \text{WriteTr}, key, obj, val) \equiv \text{<previous definition>}
\]

\[
\text{Return}(id, \text{WriteTr}, key, obj, val, \text{OK}) \equiv \text{<previous definition>} \land \text{locked}(key, obj) \\
\land \text{localvalue}(obj, key)' = \text{val}
\]

\[
\text{Return}(id, \text{WriteTr}, key, obj, val, \text{ABORT}) \equiv \\
\text{status}_U(id) = \text{(WriteTr}, key, obj, val) \land \text{Return}(key, \text{AcqLock}, obj, \text{REJECTED}) \\
\land \text{status}_U(id)' = \text{ABORTED} \\
\land H_U' = H_U @ (id, \text{AbortTr}, key) \\
\land \neg \text{allocated}(key)' \\
\land (\forall x: \text{localvalue}(x, key)' = \text{NULL})
\]

\( A_2 \) ensures that the above is a refinement.

\[
\text{Call}(id, \text{EndTr}, key) \equiv \text{<previous definition>}
\]

\[
\text{Return}(id, \text{EndTr}, key, \text{OK}) \equiv \text{<previous definition>} \land (\forall x: \text{localvalue}(x, key) = \text{NULL})
\]

\( \text{Return}(id, \text{EndTr}, key, \text{ABORT}) \) is never enabled, and is absent in the implementation.
Call (id, AbortTr, key) \iff \langle\text{previous definition}\rangle

Return (id, AbortTr, key) \iff \langle\text{previous definition}\rangle \\
\land (\forall x: \text{localvalue} (x, key) = \text{NULL})

In addition to the above refinements of the upper interface events, we define the following events. These events have null images at the upper interface because they do not update any upper interface variables.

RequestLock (id, key, obj) \iff \\
status_U (id) \in \{(\text{ReadTr}, key, obj), (\text{WriteTr}, key, obj)\} \land \neg \text{locked} (key, obj) \\
\land \text{Call} (key, \text{AcqLock}, obj)

LockAcquired (key, obj) \iff \\
\text{Return} (key, \text{AcqLock}, obj, \text{GRANTED}) \\
\land \text{locked} (key, obj)

RequestRead (id, key, obj) \iff \\
status_U (id) = (\text{ReadTr}, key, obj) \land \text{locked} (key, obj) \land \text{localvalue} (obj, key) = \text{NULL} \\
\land \text{Call} (key, \text{Read}, obj)

ReadCompleted (key, obj, val) \iff \\
\text{Return} (key, \text{Read}, obj, val) \\
\land \text{localvalue} (obj, key) = \text{val}

RequestWrite (id, key, obj) \iff \\
status_U (id) = (\text{EndTr}, key) \land \text{localvalue} (obj, key) \neq \text{NULL} \\
\land \text{Call} (key, \text{Write}, obj, \text{localvalue} (obj, key))

WriteCompleted (key, obj) \iff \\
\text{Return} (key, \text{Write}, obj, val) \\
\land \text{localvalue} (obj, key) = \text{NULL}

ReqRelLock (key, obj) \iff \\
\neg \text{allocated} (key) \land \text{locked} (key, obj) \\
\land \text{Call} (key, \text{RelLock}, obj)

LockReleased (key, obj) \iff \\
\text{Return} (key, \text{RelLock}, obj) \\
\land \neg \text{locked} (key, obj)

5. **Verification**

We first prove that \(A_1\) and \(A_2\) are invariant, thereby establishing the implementation events to be refinements of the upper interface events. We then prove the safety requirement (that
dependacy is acyclic) and the progress requirement (that every call eventually returns) in the upper interface specification.

Given a predicate \( A \) in the state variables \( \{v_i\} \), we use \( A' \) to denote \( A \) with every free occurrence of \( v_i \) replaced by \( v_i' \). We say that \( A \) is invariant given \( B \) if the following are logically valid: (i) \( \text{Initial} \Rightarrow A \); and (ii) \( A \land B \land e \Rightarrow A' \) for every event \( e \) [4]. \( \text{Initial} \) is a predicate specifying initial conditions on the state variables. \( B \) is a safety property that is either assumed, as in the case of \text{correctkeyuse} , or has been proved to be invariant separately, as in the case of legal (id).

5.1. Proof of refinement

In order to establish \( A_1 \) and \( A_2 \), we need additional safety requirements. The following requirements specify that every allocated key is associated with a unique active transaction:

\[
\begin{align*}
B_1 &\equiv \neg \text{allocated} (key) \Rightarrow (\forall id : \text{keyof} (id) \neq key) \\
B_2 &\equiv \text{allocated} (key) \Rightarrow (\exists \text{exactly one id : keyof} (id) = key)
\end{align*}
\]

Lemma 1. \( B_1 \land B_2 \) is invariant, given \text{correctkeyuse} . (Proof omitted.)

The following assertions specify relationships between state variables during the growing phase of a transaction, during which it acquires a key and then locks:

\[
\begin{align*}
C_1 &\equiv \text{status}_U (id) \in \{\text{NOTBEGIN,(BeginTr)}\} \Rightarrow (id) \notin H_U \\
C_2 &\equiv \text{keyof} (id) = key \land \text{status}_U (id) = \text{READY} \Rightarrow \text{status}_L (key) = \text{READY} \\
C_3 &\equiv \text{keyof} (id) = key \land \neg \text{locked} (key, obj) \Rightarrow (id, obj) \notin H_U \\
\quad \text{The consequent of} \ C_3 \text{ implies} \ \text{currentvalue} (obj, key) = \text{committedvalue} (obj). \\
C_4 &\equiv \text{keyof} (id) = key \land \text{status}_L (key) = (\text{AcqLock}, obj) \\
\quad \Rightarrow \neg \text{locked} (key, obj) \land \text{status}_U (id) = (key, obj) \\
C_5 &\equiv \text{keyof} (id) = key \land \text{locked} (key, obj) \land \text{status}_U (id) \neq (\text{EndTr}) \\
\quad \Rightarrow \text{storedvalue} (obj) = \text{committedvalue} (obj) \\
C_6 &\equiv \text{keyof} (id) = key \land \text{status}_U (id) = (key, obj) \land \text{locked} (key, obj) \\
\quad \land \text{localvalue} (obj, key) = \text{NULL} \Rightarrow (id, obj) \notin H_U \\
C_7 &\equiv \text{keyof} (id) = key \land \text{status}_L (key) = (\text{Read}, obj) \Rightarrow \text{locked} (key, obj) \\
\quad \land \text{localvalue} (obj, key) = \text{NULL} \land (id, obj) \notin H_U \land \text{status}_U (id) = (\text{ReadTr}, key, obj) \\
C_8 &\equiv \text{keyof} (id) = key \land \text{locked} (key, obj) \Rightarrow \exists obj \in \text{accessed} (id) \\
A_1 &\equiv \text{keyof} (id) = key \land \text{localvalue} (obj, key) \neq \text{NULL} \\
\quad \Rightarrow \text{localvalue} (obj, key) = \text{currentvalue} (obj, id)
\end{align*}
\]

where we have repeated \( A_1 \) for convenience of reference.

The following assertions specify relationships when a transaction is committing its writes:

\[
\begin{align*}
D_1 &\equiv \text{keyof} (id) = key \land \text{locked} (key, obj) \land \text{status}_U (id) = (\text{EndTr}) \land \text{localvalue} (obj, key) \neq \text{NULL} \\
\quad \Rightarrow \text{storedvalue} (obj) = \text{committedvalue} (obj) \\
D_2 &\equiv \text{keyof} (id) = key \land \text{status}_L (key) = (\text{Write}, obj, val) \\
\quad \Rightarrow \text{status}_U (id) = (\text{EndTr}, key) \land \text{val} = \text{currentvalue} (obj, id) \land \text{locked} (key, obj) \\
D_3 &\equiv \text{keyof} (id) = key \land \text{locked} (key, obj) \land \text{status}_U (id) = (\text{EndTr}) \land \text{localvalue} (obj, key) = \text{NULL} \\
\quad \Rightarrow \text{storedvalue} (obj) = \text{currentvalue} (obj, id)
\end{align*}
\]
The following assertions specify relationships during the lock releasing phase of a transaction:

\[ E_1 \equiv \neg \text{allocated} \left( \text{key} \right) \Rightarrow \text{localvalue} \left( \text{obj}, \text{key} \right) = \text{NULL} \]
\[ E_2 \equiv \text{status}_a \left( \text{key} \right) = \left( \text{RelLock}, \text{obj} \right) \Rightarrow \neg \text{allocated} \left( \text{key} \right) \land \text{locked} \left( \text{key}, \text{obj} \right) \]
\[ E_3 \equiv \neg \text{locked} \left( \text{key}, \text{obj} \right) \Rightarrow \text{localvalue} \left( \text{obj}, \text{key} \right) = \text{NULL} \]

Two additional assertions are needed:

\[ F_1 \equiv \left( \forall \text{key} : \neg \text{locked} \left( \text{key}, \text{obj} \right) \right) \Rightarrow \text{storedvalue} \left( \text{obj} \right) = \text{committedvalue} \left( \text{obj} \right) \]
\[ F_2 \equiv \text{owned} \left( \text{key}, \text{obj} \right) \iff \text{locked} \left( \text{key}, \text{obj} \right) \]

We use the notation \( C_{1-8} \) to denote the conjunction \( C_1 \land C_2 \cdots \land C_8 \).

**Lemma 2.** \( A_1 \land C_{1-8} \land D_{1-3} \land E_{1-3} \land F_{1-2} \) is invariant, given \( B_1 \land B_2 \). (Proof omitted.)

Lemma 2 establishes \( A_1 \). We next show that it also establishes \( A_2 \). Assume \text{deadlock} \left( \text{key}, \text{obj} \right) \) to be true. From the definition of \text{deadlock}, we have \text{waiting} \left( \text{key}, \text{obj} \right) \) and \text{owned} \left( k, \text{obj} \right) \) for some \( k \neq \text{key} \). From the definition of \text{waiting} \) and \( C_4 \), \text{waiting} \left( \text{key}, \text{obj} \right) \) implies \text{status}_U \left( \text{id} \right) = \left( \text{obj} \right) \) for some \( \text{id} \), which implies \( \text{obj} \in \text{accessed} \left( \text{id} \right) \). From \( F_2 \) and \( C_8 \), \text{owned} \left( k, \text{obj} \right) \) implies \( \text{obj} \in \text{accessed} \left( \text{id}_i \right) \) for some \( \text{id}_i \). From \( B_1 \land B_2 \), we have \( \text{id}_1 \neq \text{id} \). Consequently, \text{concurrentaccess} \left( \text{id} \right) \) is true.

### 5.2. Proof of serializability

The following assertions are to be proved:

\[ G_1 \equiv \left( \text{id}_1, \text{obj} \right) \in \text{H}_U \land \left( \text{id}_1, \text{EndTr} \right) \notin \text{H}_U \land \left( \text{id}_1, \text{AbortTr} \right) \notin \text{H}_U \]
\[ \Rightarrow \text{keyof} \left( \text{id}_1 \right) \neq \text{NULL} \land \text{locked} \left( \text{keyof} \left( \text{id}_1 \right), \text{obj} \right) \]
\[ G_2 \equiv \left( \text{id}_1, \text{obj} \right) \circ \left( \text{id}_2, \text{obj} \right) \) is a subsequence of \( \text{H}_U \)
\[ \Rightarrow \left( \text{id}_1, \text{obj} \right) \circ \left( \text{id}_1, \text{EndTr} \right) \circ \left( \text{id}_2, \text{obj} \right) \) is a subsequence of \( \text{H}_U \)
\[ \land \left( \text{id}_1, \text{obj} \right) \circ \left( \text{id}_1, \text{AbortTr} \right) \circ \left( \text{id}_2, \text{obj} \right) \) is a subsequence of \( \text{H}_U \)

**Lemma 3.** \( G_1 \land G_2 \) is invariant, given \( B_1 \land B_2 \land \text{legal} \left( \text{id}_1 \right) \). (Proof omitted.)

Lemma 3 implies that the following is invariant:

\[ G_3 \equiv \text{dependency} \left( \text{id}_1, \text{id}_2 \right) \Rightarrow \left( \text{id}_1, \text{EndTr} \right) \circ \left( \text{id}_2, \text{EndTr} \right) \) is a subsequence of \( \text{C} \left( \text{H}_U \right) \)

This can be proved as follows. Given \text{dependency} \left( \text{id}_1, \text{id}_2 \right) \), from the definitions of \( \text{C} \left( \text{H}_U \right) \) and \( \text{legal} \left( \text{id}_2 \right) \), \( \left( \text{id}_2, \text{obj} \right) \) \in \( \text{C} \left( \text{H}_U \right) \) implies \( \left( \text{id}_2, \text{obj} \right) \circ \left( \text{id}_2, \text{EndTr} \right) \) is a subsequence of \( \text{C} \left( \text{H}_U \right) \). Combining this with \( G_2 \), we see that \( \left( \text{id}_1, \text{obj} \right) \circ \left( \text{id}_2, \text{obj} \right) \) is a subsequence of \( \text{C} \left( \text{H}_U \right) \) implies \( \left( \text{id}_1, \text{obj} \right) \circ \left( \text{id}_1, \text{EndTr} \right) \circ \left( \text{id}_2, \text{obj} \right) \circ \left( \text{id}_2, \text{EndTr} \right) \) is a subsequence of \( \text{C} \left( \text{H}_U \right) \).

\( G_3 \) implies that \text{dependency} \) is acyclic. Assume the contrary: For some \( n \geq 2 \), there exist distinct \( \text{id}_1, \text{id}_2, \ldots, \text{id}_n \), such that \text{dependency} \left( \text{id}_k, \text{id}_{k+1} \right) \) for \( k=1, \ldots, n-1 \), and \text{dependency} \left( \text{id}_n, \text{id}_1 \right) \). From \( G_3 \), \( \text{C} \left( \text{H}_U \right) \) contains the subsequences \( \left( \text{id}_k, \text{EndTr} \right) \circ \left( \text{id}_{k+1}, \text{EndTr} \right) \), for \( k=1, \ldots, n-1 \), and \( \left( \text{id}_n, \text{EndTr} \right) \circ \left( \text{id}_1, \text{EndTr} \right) \). But this implies that there are at least two occurrences of \( \left( \text{id}_1, \text{EndTr} \right) \) in \( \text{C} \left( \text{H}_U \right) \), which violates \text{legal} \left( \text{id}_1 \right).

### 5.3. Proof of progress

Given predicates \( A \) and \( B \) and an event \( \epsilon_1 \), we say that \( A \) \text{ leads to } \( B \) \text{ via } \epsilon_1 \) if the following are logically valid: (i) \( A \Rightarrow \text{enabled} \left( \epsilon_1 \right) \), (ii) \( A \land \epsilon_1 \Rightarrow B' \), and (iii) \( A \land \epsilon \Rightarrow A' \lor B' \) for every event \( \epsilon \) \text{ [4]}. Whenever \( A \) holds, parts (i) and (ii) ensure that \( \epsilon_1 \) is enabled and its occurrence makes \( B \) hold. Part (iii) ensures that no event can violate \( A \) without establishing \( B \). Thus, in any fair implementation \( B \) will hold at some point. We use \text{leads to} \) to denote the
closure of the \textit{leads-to-via} relation; e.g., \( A \text{ leads-to } B \) if \( A \text{ leads-to } B \lor C \) and \( C \text{ leads-to } B \).

We first show that the progress properties \( Q_1 \), \( Q_2 \), \( Q_3 \), and \( Q_4 \) offered by the lower interface can be assumed in the verification. Each such property has the form \( A \text{ leads-to } B \) and is achieved by the execution of a lower interface event \( e_L \). Let \( e_L \) be imbedded together with enabling condition \( b \) in event \( e_I \) of the implementation. In order to assume the property \( A \text{ leads-to } B \), we need to establish that \( A \implies b \) is invariant.

Consider \( Q_1 \), which is achieved by executing the \( \text{Return} (\text{key,Read, } obj, val) \) event. This event is imbedded in the implementation event \( \text{ReadCompleted} (\text{key, obj, val}) \). The latter event is enabled whenever the former is enabled, because it has no other requirement in its enabling condition. Consequently, \( Q_1 \) can be assumed.

Similarly, we can assume \( Q_2 \) because \( \text{WriteCompleted} (\text{key, obj}) \) is enabled whenever \( \text{Return} (\text{key, Write, obj, val}) \) is enabled.

We can assume \( Q_3 \) because \( \text{LockReleased} (\text{key, obj}) \) is enabled whenever \( \text{Return} (\text{key, Relock, obj}) \) is enabled.

We can assume \( Q_4 \) because (i) \( \text{LockAcquired} \) is enabled whenever \( \text{Return} (\text{AcqLock, GRANTED}) \) is enabled; and (ii) either \( \text{Return} (\text{ReadTr, ABORT}) \) or \( \text{Return} (\text{WriteTr, ABORT}) \) is continuously enabled whenever \( \text{Return} (\text{AcqLock, REJECTED}) \) is enabled. Part (ii) holds because of \( C_4 \).

From \( Q_3 \), \( \text{ReqRelock} \), and \( \text{LockReleased} \), we can establish:
\[
\text{holdinglocks (key)} \land \neg \text{allocated (key)} \text{ leads-to } \neg \text{holdinglocks (key)}
\]

From this and \( \text{Return} (\text{BeginTr}) \), we can establish:

\[ W_1 \equiv \text{status}_U (id) = (\text{BeginTr}) \text{ leads-to } \text{status}_U (id) \in \{ \text{READY, NOTBEGUN} \} \]

From \( Q_2 \), \( \text{ReqWrite} \), and \( \text{WriteCompleted} \), we have:
\[
\text{status}_U (id) = (\text{EndTr, key}) \land \text{localvalue (obj, key)} \neq \text{NULL} \\
\text{leads-to } \text{status}_U (id) = (\text{EndTr, key}) \land \text{localvalue (obj, key)} = \text{NULL}
\]

From \( \text{Return} (\text{EndTr, key}) \), we have:
\[
\text{status}_U (id) = (\text{EndTr, key}) \land \text{localvalue (obj, key)} = \text{NULL} \\
\text{leads-to } \text{status}_U (id) = \text{COMMITTED}
\]

Combining the above two, we have:

\[ W_2 \equiv \text{status}_U (id) = (\text{EndTr, key}) \text{ leads-to } \text{status}_U (id) = \text{COMMITTED} \]

From \( \text{Return} (\text{AbortTr}) \), we have:

\[ W_3 \equiv \text{status}_U (id) = (\text{AbortTr}) \text{ leads-to } \text{status}_U (id) = \text{ABORTED} \]

From \( Q_1 \), \( \text{ReqRead} \) and \( \text{ReadCompleted} \), we have:
\[
\text{status}_U (id) = (\text{ReadTr, key, obj}) \land \text{locked (key, obj)} \\
\text{leads-to } \text{status}_U (id) = (\text{ReadTr, key, obj}) \land \text{localvalue (obj, key)} \neq \text{NULL}
\]

From above and \( \text{Return} (\text{ReadTr, val}) \), we have:

\[ W_4 \equiv \text{status}_U (id) = (\text{ReadTr, key, obj}) \land \text{locked (key, obj)} \text{ leads-to } \text{status}_U (id) = \text{READY} \]

From \( \text{Return} (\text{WriteTr, OK}) \), we have:

\[ W_5 \equiv \text{status}_U (id) = (\text{WriteTr, key, obj}) \land \text{locked (key, obj)} \text{ leads-to } \text{status}_U (id) = \text{READY} \]
From RequestLock, we have:
\[ W_6 \equiv \text{status}_\mathcal{U}(id) = (\text{key}, \text{obj}) \land \neg \text{locked}(\text{key}, \text{obj}) \text{ leads to } \text{waiting}(\text{key}, \text{obj}) \]

From \( W_1, W_2, W_3, W_4, W_5, \) and \( W_6, \) all that is left to establish the desired progress property is:
\[ \text{status}_\mathcal{U}(id) = (\text{key}, \text{obj}) \land \text{waiting}(\text{key}, \text{obj}) \text{ leads to } \neg \text{waiting}(\text{key}, \text{obj}). \]

Observe that this is the same as \( G_4, \) the consequent of \( Q_4. \) We now provide a proof of this.

**Lexicographic induction**

Consider the directed graph of nodes \( \text{KEYS} \cup \text{OBJECTS}, \) and edges \( \{(x, k) : \text{owned}(k, x)\} \cup \{(k, x) : \text{waiting}(k, x)\}. \) Consider any key \( k_1 \) waiting on object \( x_1. \) We need to show that eventually \( \neg \text{waiting}(k_1, x_1) \) holds. Let \( M = | \text{KEYS} | \)

Each node in this graph can have several incoming edges, but at most one outgoing edge. We say \( k_1, x_1, k_2, x_2, \ldots, k_j \) is a path if \( \text{waiting}(k_i, x_i) \) and \( \text{owned}(k_{i+1}, x_i) \) for \( 1 \leq i < j, \) and all the \( k_i \)’s and \( x_i \)’s are distinct. We say that \( x_i \) is not locked if \( \forall \text{key} : \neg \text{locked}(\text{key}, x_i). \) We say that \( k_i \) is not waiting if \( \forall \text{obj} : \neg \text{waiting}(k_i, \text{obj}). \)

Define the following state functions on the directed graph, where \( 1 \leq j \leq M : \)

\[
\text{waitstate}_1(j) : \text{boolean} \\
\quad \text{True iff there are } k_2, x_2, \ldots, k_j \text{ such that } k_1, x_1, \ldots, k_j \text{ is a path and } k_j \text{ is not waiting.}
\]

\[
\text{waitstate}_2(j) : \text{boolean} \\
\quad \text{True iff there are } k_2, x_2, \ldots, k_j, x_j \text{ such that } k_1, x_2, \ldots, k_j \text{ is a path, and} \\
\quad \text{waiting}(k_j, x_j) \text{ and } x_j \text{ is not locked.}
\]

\[
\text{waitstate}_3(j) : \text{boolean} \\
\quad \text{True iff there are } k_2, x_2, \ldots, k_j \text{ such that } k_1, x_2, \ldots, k_j \text{ is a path, and} \\
\quad \text{waiting}(k_j, x_i) \text{ for some } x_i \text{ such that } 1 \leq i < j \text{ or owned}(k_1, x_i); \text{i.e., } k_j \text{ is deadlocked.}
\]

Observe that for any state of the directed graph, exactly one of the \( 3M \) functions \( \{\text{waitstate}_1(j), \text{waitstate}_2(j), \text{waitstate}_3(j) : 1 \leq j \leq M \} \) is true. At any time, let the function \( \text{depth} \) denote that value of \( j. \)

Define the following functions, where \( 1 \leq i \leq M : \)

\[
\beta(i) : \text{integer} \\
\quad \text{If } i < \text{depth, or if } i = \text{depth} \text{ and } k_i \text{ is waiting: } \beta(i) \text{ equals the number of times } x_i \text{ has} \\
\quad \text{been unlocked since the last time that } k_i \text{ started to wait.} \\
\quad \text{If } i = \text{depth} \text{ and } k_i \text{ is not waiting: } \beta(i) = 0. \\
\quad \text{If } i > \text{depth} : \beta(i) = -1.
\]

\[
\alpha(i) : \text{integer} \\
\quad \text{If } i < \text{depth, or if } i = \text{depth} \text{ and } k_i \text{ is not releasing locks (i.e., allocated}(k_i) \text{ is true): } \alpha(i) \text{ equals the number of objects held by } k_i. \\
\quad \text{If } i = \text{depth} \text{ and } k_i \text{ is releasing locks: } \alpha(i) \text{ equals the number of objects locked by } k_i \text{ just} \\
\quad \text{before it started releasing locks.} \\
\quad \text{If } i > \text{depth} : \alpha(i) = 0.
\]

Define the function \( \alpha = (\beta(1), \alpha(2), \alpha(3), \ldots, \alpha(M), \beta(M)) \). The values of \( \alpha \) can be well-ordered using the lexicographic ordering. We will show that \( \alpha \) increases without bound unless \( \neg \text{waiting}(k_1, x_1) \) becomes true. This establishes the desired progress property as follows: \( \alpha \)
increasing without bound implies that either $\beta(i)$ or $\alpha(i)$ increases without bound for some $i$. The former is not allowed by the fairness assumption of the lock manager (i.e., $Q_4$). The latter is not allowed by the assumption that every transaction needs at most a finite set of locks (i.e., $L_2$).

Let $\alpha$ have the value $a = (b_1, a_2, b_2, a_3, \ldots, a_M, b_M)$. The notation $a = a[\beta(i) = z]$ means that every function in $\alpha$ has the same value as in $a$ except for $\beta(i)$ which has the value $x$. Other relations can be used in place of equality; e.g., $\alpha = a[\beta(i) \geq 0]$. This notation is extended in the usual fashion: e.g., $\alpha = a[\alpha(i) = x; \beta(j) = y]$; $\alpha = a[\alpha(i) = x; j \leq i \leq k]$.

The following leads-to properties can be established.

$$W_7 \equiv \text{waitstate}_1(j) \land \alpha = a \text{ leads-to } W_{7a} \lor W_{7b} \lor W_{7c} \lor W_{7d},$$

where

$$W_{7a} \equiv \text{waitstate}_1(j-1) \land \alpha = a[\beta(j-1) = b_{j-1} + 1; \alpha(j) = 0; \beta(j) = 1] > a$$

$$W_{7b} \equiv \text{waitstate}_1(k) \land j < k \land \alpha = a[\beta(i) = 0; \alpha(i) \geq 0; j < i \leq k] > a$$

$$W_{7c} \equiv \text{waitstate}_1(k) \land j \leq k \land \alpha = a[\beta(i) = 0; \alpha(i) \geq 0; j < i \leq k] > a$$

$$W_{7d} \equiv \text{waitstate}_1(k) \land j \leq k \land \alpha = a[\beta(i) = 0; \alpha(i) \geq 0; j < i \leq k] > a$$

$W_{7a}$ results if $k_j$ returns the lock on $x_{j-1}$ in the process of releasing its locks. The value of $\alpha$ increases because $\beta(j-1)$ increases and it is lexicographically the most significant of the functions whose values are changed. $W_{7c}$ with $j = k$ results if $k_j$ requests an object that is not locked. $W_{7b} \lor W_{7c} \land j < k$ results if the object is already locked but not by any $k_i$, $i < j$. $W_{7d}$ results if the object is already locked by some $k_i$ where $i < j$. In $W_{7b} \lor W_{7c} \lor W_{7d}$, if $k > j$, the value of $\alpha$ increases because $\beta(j+1)$ increases from $-1$ to $0$, $\alpha(j+1)$ stays at $0$ or increases, and the other functions whose values are changed are less significant. If $k = j$, $\alpha$ stays constant. One of the above transitions will eventually occur because $k_j$ is ready in $\text{waitstate}_1(j)$, i.e., status$^\text{op}(id) = \text{READY}$ where keyof (id) = $k_j$.

$$W_8 \equiv \text{waitstate}_2(j) \land \alpha = a \text{ leads-to } \neg \text{waiting}(k_1, x_1) \lor W_{8a} \lor W_{8b},$$

where

$$W_{8a} \equiv \text{waitstate}_1(j) \land \alpha = a[\alpha(j) = q_j + 1; \beta(j) = 0] > a$$

$$W_{8b} \equiv \text{waitstate}_1(j+1) \land \alpha = a[\beta(j+1) = 0; \alpha(j+1) \geq 0] > a$$

The LockAcquired($k_j, x_j$) event is continuously enabled in $\text{waitstate}_2(j)$. Its occurrence results in $\neg \text{waiting}(k_1, x_1)$ if $j = 1$, and in $W_{8a}$ if $j > 1$. This will eventually occur unless $x_j$ is locked by a key other than $k_j$. In this case, that key becomes $k_{j+1}$ and $W_{8b}$ holds. In the case of $W_{8a}$, the value of $\alpha$ increases because $\alpha(j)$ increases and it is the most significant function changed. In the case of $W_{8b}$, the value of $\alpha$ increases because $\beta(j+1)$ increases from $-1$ to $0$, $\alpha(j+1)$ stays at 0 or increases, and no other functions change values.

$$W_9 \equiv \text{waitstate}_3(j) \land \alpha = a \text{ leads-to } \neg \text{waiting}(k_1, x_1) \lor W_{9a},$$

where

$$W_{9a} \equiv \text{waitstate}_3(k) \land j < k \land \alpha = a[\beta(k) = b_k + 1; \beta(i) = 0; \alpha(i) = -1; k < i \leq j] > a$$

$\text{waitstate}_3(j)$ implies a cycle involving $k_j$. This cycle only involves keys from $k_1, k_2, \ldots, k_j$. LockRejected($k_i, x_j$) for every $k_i$ involved in the cycle is enabled, and the lock manager will execute one of them eventually. $\neg \text{waiting}(k_1, x_1)$ results if $k_1$ is involved in the deadlock and LockRejected($k_k, x_k$) occurs. If LockRejected($k_{k+1}, x_{k+1}$) occurs, then $k_{k+1}$ is aborted, and it gives up its locks. $W_{9a}$ results when it gives up its lock on $x_k$. At this point, the value of $\alpha$ increases because $\beta(k)$ increases, and all other function changes are less significant.

Substituting $W_8$ and $W_9$ for $W_{7c}$ and $W_{7d}$, we have $\text{waitstate}_1(j) \land \alpha = a \text{ leads-to } \alpha > a \lor \neg \text{waiting}(k_1, x_1)$. From $W_8$, we have $\text{waitstate}_2(j) \land \alpha = a \text{ leads-to } \alpha > a \lor \neg \text{waiting}(k_1, x_1)$. From $W_9$, we have $\text{waitstate}_3(j) \land \alpha = a \text{ leads-to } \alpha > a$. 


\( \forall \neg \text{waiting}(k_1, x_1) \). Combining these three \textit{leads-to} statements, we have \( \alpha \equiv \text{a. leads-to} \alpha \to \text{a. waiting}(k_1, x_1) \), which establishes that \( \alpha \) increases without bound unless \( k_1 \) stops waiting.

**5.4. Conclusion**

We have provided the sketch of a proof that the two-phase locking implementation satisfies the upper interface specification, assuming that the lower interface specification holds, as follows:

(i) Implementation events are refinements of the upper interface events; thus, safety properties of the event-driven system in the upper interface specification are also safety properties of the implementation.

(ii) The implementation satisfies the safety and progress requirements in the upper interface specification.

**REFERENCES**


