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

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.

client programs

____________________ upper interface

two-phase locking system

____________________ lower interface

physical database

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_i is the image of s_i. 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_1 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 $e_1 \equiv (v_2<5 \wedge v_1' = v_1+1)$.

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, $e_1, e_2, \cdots, e_n$, if $e_I \Rightarrow e_1 \vee e_2 \vee \cdots \vee e_n$. Given the safety requirement S, e_I is a refinement if $S \wedge e_I \Rightarrow e_1 \vee e_2 \vee \cdots \vee e_n$.

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 $S \wedge e_I \Rightarrow e_U$.

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 $obj \in$ OBJECTS, let its initial value be given by 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 *outstanding* if it has called the procedure and not yet returned. We say that the transaction is *active* if it has returned from a *BeginTr* call with a key, and it has not yet ended.

2.1. State variables

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

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

$status_U(id)$: $\{\text{NOTBEGUN, READY, COMMITTED, ABORTED}\} \cup \{(BeginTr), (ReadTr, key, obj), (WriteTr, key, obj, val), (EndTr, key), (AbortTr, key)\}$.
Initially, $status_U(id) =$ NOTBEGUN.

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

$allocated(key)$: boolean. Initially false.
True iff key is allocated to a transaction.

When we refer to a tuple in the domain of $status_U(id)$, such as $(ReadTr, key, obj)$, where a component in the tuple can have any of its allowed values, we shall omit that component in our reference. For example, $status_U(id) = (ReadTr, obj)$ means $status_U(id) = (ReadTr, key, obj)$ for some value of key. More than one component in a tuple may be omitted. For example, (obj) refers to $(ReadTr, key, obj)$ for some key or $(WriteTr, key, obj, val)$ 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 $(id, ReadTr, obj, key, val)$ or a $(id, WriteTr, obj, key, val, \text{OK})$ entry for some key and some val.

2.2. State functions

$active(id)$: boolean
True iff $(id, BeginTr) \in H_U$, and neither $(id, EndTr)$ nor $(id, AbortTr)$ is in H_U.

$accessed(id)$: powerset of OBJECTS
The set of objects that have been accessed by an active transaction id.
$=$ empty, if $\neg active(id)$.

$= \{ obj : status_U(id) = (obj) \vee (id, obj) \in H_U \}$, if $active(id)$.

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

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

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

$Call(id, BeginTr) \equiv$
$status_U(id)=$NOTBEGUN
$\wedge\ status_U(id)' = (BeginTr)$

$Return(id, BeginTr, key) \equiv$
$status_U(id) = (BeginTr) \wedge \neg allocated(key)$
$\wedge\ status_U(id)' =$READY
$\wedge\ allocated(key)'$
$\wedge\ H_U' = H_U @ (id, BeginTr, key)$

$Return(id, BeginTr, \text{FAILED}) \equiv$
$status_U(id) = (BeginTr) \wedge (\forall key : allocated(key))$
$\wedge\ status_U(id)' =$NOTBEGUN

$Call(id, ReadTr, key, obj) \equiv$
$status_U(id)=$READY $\wedge\ allocated(key)$
$\wedge\ status_U(id)' = (ReadTr, key, obj)$

$Return(id, ReadTr, key, obj, val) \equiv$
$status_U(id) = (ReadTr, key, obj)$
$\wedge\ status_U(id)' =$READY
$\wedge\ val = currentvalue(obj, id)$
$\wedge\ H_U' = H_U @ (id, ReadTr, key, obj, val)$

$$Return(id, ReadTr, key, obj, \mathrm{ABORT}) \equiv$$
$$status_U(id) = (ReadTr, key, obj) \wedge concurrentaccess(id)$$
$$\wedge\ status_U(id)' = \mathrm{ABORTED}$$
$$\wedge\ \neg allocated(key)'$$
$$\wedge\ H_U{}' = H_U @ (id, AbortTr, key)$$

$$Call(id, WriteTr, key, obj, val) \equiv$$
$$status_U(id) = \mathrm{READY} \wedge allocated(key)$$
$$\wedge\ status_U(id)' = (WriteTr, key, obj, val)$$

$$Return(id, WriteTr, key, obj, val, \mathrm{OK}) \equiv$$
$$status_U(id) = (WriteTr, key, obj, val)$$
$$\wedge\ status_U(id)' = \mathrm{READY}$$
$$\wedge\ H_U{}' = H_U @ (id, WriteTr, key, obj, val, \mathrm{OK})$$

$$Return(id, WriteTr, key, obj, val, \mathrm{ABORT}) \equiv$$
$$status_U(id) = (WriteTr, key, obj, val) \wedge concurrentaccess(id)$$
$$\wedge\ status_U(id)' = \mathrm{ABORTED}$$
$$\wedge\ \neg allocated(key)'$$
$$\wedge\ H_U{}' = H_U @ (id, AbortTr, key)$$

$$Call(id, EndTr, key) \equiv$$
$$status_U(id) = \mathrm{READY} \wedge allocated(key)$$
$$\wedge\ status_U(id)' = (EndTr, key)$$

$$Return(id, EndTr, key, \mathrm{OK}) \equiv$$
$$status_U(id) = (EndTr, key)$$
$$\wedge\ status_U(id)' = \mathrm{COMMITTED}$$
$$\wedge\ \neg allocated(key)'$$
$$\wedge\ H_U{}' = H_U @ (id, EndTr, key, \mathrm{OK})$$

$$Return(id, EndTr, key, \mathrm{ABORT}) \equiv$$
$$status_U(id) = (EndTr, key) \wedge concurrentaccess(id)$$
$$\wedge\ status_U(id)' = \mathrm{ABORTED}$$
$$\wedge\ \neg allocated(key)'$$
$$\wedge\ H_U{}' = H_U @ (id, AbortTr, key)$$

$$Call(id, AbortTr, key) \equiv$$
$$status_U(id) = \mathrm{READY} \wedge allocated(key)$$
$$\wedge\ status_U(id)' = (AbortTr, key)$$

$$Return(id, AbortTr, key) \equiv$$
$$status_U(id) = (AbortTr, key)$$
$$\wedge\ status_U(id)' = \mathrm{ABORTED}$$
$$\wedge\ \neg allocated(key)'$$
$$\wedge\ H_U{}' = H_U @ (id, AbortTr, key)$$

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:

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

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

An invocation of *ReadTr* , *WriteTr* , 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 *concurrentaccess* (*id*) in the enabling conditions of the corresponding return events.

The definition of *committedvalue* ensures that writes of aborted transactions do not influence the committed values. The definition of *currentvalue* ensures that the values read by a transaction are not affected by writes of other concurrently executing transactions. Observe that *currentvalue* (obj, id_1) can differ from *currentvalue* (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 $status_U(id)$=COMMITTED.

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

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

Define the following state functions:

keyof (*id*): KEYS ∪ {NULL}
=NULL, if ¬*active* (*id*).
=*key* , if *active* (*id*) and (*id* ,*BeginTr* ,*key*) in H_U.

correctkeyuse : boolean.
True iff every transaction has used the correct key in all its procedure calls, i.e., every $(id, key) \in H_U$ satisfies *key* =*keyof* (*id*).

The upper interface specification includes the following:

Safety requirement: *correctkeyuse* ⟹ *dependency* is acyclic.

2.5. Progress requirements

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

$$L_1 \equiv status_U(id) \in \{(BeginTr), (ReadTr), (WriteTr), (EndTr), (AbortTr)\}$$
$$\textit{leads-to } status_U(id) \in \{\text{READY, ABORTED, COMMITTED, NOTBEGUN}\}$$

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

$$L_2 \equiv (status_U(id) \in \{(ReadTr),(WriteTr)\}\ leads\text{-}to\ status_U(id)=\text{READY})$$
$$\Longrightarrow (status_U(id)=\text{READY}\ leads\text{-}to\ status_U(id)=(EndTr))$$

The upper interface specification includes the following:

Progress requirement: $correctkeyuse \wedge L_2 \Longrightarrow L_1$.

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

$status_L(key)$: {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.

$owned(key, obj)$: boolean. Initially false.

True iff *key* has locked *obj*.

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

The value of the object in the physical database.

3.2. State functions

$waiting(key, obj)$: boolean.

True iff $status_L(key)=(AcqLock, obj)$. Defined for notational convenience.

$deadlock(key, obj)$: 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*.

$$Call(key, AcqLock, obj) \equiv$$
$$status_L(key)=\text{READY}$$
$$\wedge\ status_L(key)' = (AcqLock, obj)$$

$$Return(key, AcqLock, obj, \text{GRANTED}) \equiv$$
$$status_L(key)=(AcqLock, obj) \wedge (\forall k: \neg owned(k, obj))$$
$$\wedge\ status_L(key)' = \text{READY}$$
$$\wedge\ owned(key, obj)'$$

$Return(key, AcqLock, obj, \text{REJECTED}) \equiv$
$\quad status_L(key) = (AcqLock, obj) \wedge deadlock(key, obj)$
$\quad \wedge\ status_L(key)' = \text{READY}$

$Call(key, RelLock, obj) \equiv$
$\quad status_L(key) = \text{READY}$
$\quad \wedge\ status_L(key)' = (RelLock, obj)$

$Return(key, RelLock, obj) \equiv$
$\quad status_L(key) = (RelLock, obj) \wedge owned(key, obj)$
$\quad \wedge\ status_L(key)' = \text{READY}$
$\quad \wedge\ \neg owned(key, obj)'$

$Call(key, Read, obj) \equiv$
$\quad status_L(key) = \text{READY}$
$\quad \wedge\ status_L(key)' = (Read, obj)$

$Return(key, Read, obj, val) \equiv$
$\quad status_L(key) = (Read, obj)$
$\quad \wedge\ status_L(key)' = \text{READY}$
$\quad \wedge\ val = storedvalue(obj)$

$Call(key, Write, obj, val) \equiv$
$\quad status_L(key) = \text{READY}$
$\quad \wedge\ status_L(key)' = (Write, obj, val)$

$Return(key, Write, obj, val) \equiv$
$\quad status_L(key) = (Write, obj, val)$
$\quad \wedge\ status_L(key)' = \text{READY}$
$\quad \wedge\ storedvalue(obj)' = val$

3.4. Safety requirements

The enabling condition of $Return(key, AcqLock, obj, \text{GRANTED})$ 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 $Return(key, AcqLock, obj, \text{REJECTED})$ ensures that (key, obj) is involved in a deadlock.

3.5. Progress requirements

The lower layer guarantees the progress properties Q_1 through Q_4:

$Q_1 \equiv status_L(key) = (Read)\ leads\text{-}to\ status_L(key) = \text{READY}$

$Q_2 \equiv status_L(key) = (Write)\ leads\text{-}to\ status_L(key) = \text{READY}$

$Q_3 \equiv status_L(key) = (RelLock, obj) \wedge owned(key, obj)$
$\quad leads\text{-}to\ status_L(key) = \text{READY} \wedge \neg owned(key, obj)$

$Q_4 \equiv R_4 \Rightarrow G_4$ where

$R_4 \equiv waiting(k_1, obj) \wedge owned(k_2, obj)$ *leads-to* $\neg owned(k_2, obj))$

$G_4 \equiv waiting(k_1, obj)$ *leads-to* $\neg waiting(k_1, obj)$

Q_4 specifies the property that every call to *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 (*key*, *obj*): boolean. Initially false.
Indicates whether *key* has locked *obj*.

localvalue (*obj*, *key*): VALUES ∪ {NULL}. Initially NULL.
The current value of *obj* as seen by transaction using *key*.

The upper interface variable H_U becomes auxiliary. This also makes auxiliary all state functions defined in terms of H_U, such as *concurrentaccess*, *currentvalue*, *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 (*key*): boolean.
True iff *locked* (*key*, *obj*) is true for some *obj*.

4.3. Events

Implementation events that are refinements of the upper interface events are listed first. In an event predicate, the notation <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.

Call (*id*, *BeginTr*) ≡ <previous definition>

$$
\begin{aligned}
&Return(id, BeginTr, result) \equiv \\
&\quad status_U(id) = (BeginTr) \\
&\quad \wedge ((\exists key : \neg allocated(key) \wedge \neg holdinglocks(key) \\
&\qquad \wedge\ allocated(key)' \wedge status_U(id)' = \text{READY} \wedge result = key \\
&\qquad \wedge\ H_U' = H_U @ (id, BeginTr, key)) \\
&\quad \vee ((\forall key : allocated(key)) \\
&\qquad \wedge\ status_U(id)' = \text{NOTBEGUN} \wedge result = \text{FAILED} \wedge H_U' = H_U))
\end{aligned}
$$

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

$Call(id, ReadTr, key, obj) \equiv$ <previous definition>

$$\begin{aligned} &Return(id, ReadTr, key, obj, val) \equiv \\ &\quad status_U(id)=(ReadTr, key, obj) \wedge localvalue(obj, key) \neq \text{NULL} \\ &\quad \wedge\, status_U(id)' = \text{READY} \\ &\quad \wedge\, H_U' = H_U @ (id, ReadTr, key, obj, val) \\ &\quad \wedge\, val = localvalue(obj, key) \end{aligned}$$

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

$$\begin{aligned} A_1 \equiv\ & keyof(id)=key \wedge localvalue(obj, key) \neq \text{NULL} \\ & \Rightarrow localvalue(obj, key)=currentvalue(obj, id) \end{aligned}$$

$$\begin{aligned} &Return(id, ReadTr, key, obj, \text{ABORT}) \equiv \\ &\quad status_U(id)=(ReadTr, key, obj) \wedge Return(key, AcqLock, obj, \text{REJECTED}) \\ &\quad \wedge\, status_U(id)' = \text{ABORTED} \\ &\quad \wedge\, H_U' = H_U @ (id, AbortTr, key) \\ &\quad \wedge\, \neg allocated(key)' \\ &\quad \wedge\, (\forall x : localvalue(x, key)' = \text{NULL}) \end{aligned}$$

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

$$A_2 \equiv\ keyof(id)=key \wedge deadlock(key, obj) \Rightarrow concurrentaccess(id)$$

$Call(id, WriteTr, key, obj, val) \equiv$ <previous definition>

$Return(id, WriteTr, key, obj, val, \text{OK}) \equiv$ <previous definition> $\wedge\, locked(key, obj)$
$\quad \wedge\, localvalue(obj, key)' = val$

$$\begin{aligned} &Return(id, WriteTr, key, obj, val, \text{ABORT}) \equiv \\ &\quad status_U(id)=(WriteTr, key, obj, val) \wedge Return(key, AcqLock, obj, \text{REJECTED}) \\ &\quad \wedge\, status_U(id)' = \text{ABORTED} \\ &\quad \wedge\, H_U' = H_U @ (id, AbortTr, key) \\ &\quad \wedge\, \neg allocated(key)' \\ &\quad \wedge\, (\forall x : localvalue(x, key)' = \text{NULL}) \end{aligned}$$

A_2 ensures that the above is a refinement.

$Call(id, EndTr, key) \equiv$ <previous definition>

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

$Return(id, EndTr, key, \text{ABORT})$ is never enabled, and is absent in the implementation.

$Call(id, AbortTr, key) \equiv$ <previous definition>

$Return(id, AbortTr, key) \equiv$ <previous definition>

$\wedge (\forall x : localvalue(x, key)' = \text{NULL})$

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.

$$RequestLock(id, key, obj) \equiv$$
$$status_U(id) \in \{(ReadTr, key, obj), (WriteTr, key, obj)\} \wedge \neg locked(key, obj)$$
$$\wedge\ Call(key, AcqLock, obj)$$

$$LockAcquired(key, obj) \equiv$$
$$Return(key, AcqLock, obj, \text{GRANTED})$$
$$\wedge\ locked(key, obj)'$$

$$RequestRead(id, key, obj) \equiv$$
$$status_U(id) = (ReadTr, key, obj) \wedge locked(key, obj) \wedge localvalue(obj, key) = \text{NULL}$$
$$\wedge\ Call(key, Read, obj)$$

$$ReadCompleted(key, obj, val) \equiv$$
$$Return(key, Read, obj, val)$$
$$\wedge\ localvalue(obj, key)' = val$$

$$RequestWrite(id, key, obj) \equiv$$
$$status_U(id) = (EndTr, key) \wedge localvalue(obj, key) \neq \text{NULL}$$
$$\wedge\ Call(key, Write, obj, localvalue(obj, key))$$

$$WriteCompleted(key, obj) \equiv$$
$$Return(key, Write, obj, val)$$
$$\wedge\ localvalue(obj, key)' = \text{NULL}$$

$$ReqRelLock(key, obj) \equiv$$
$$\neg allocated(key) \wedge locked(key, obj)$$
$$\wedge\ Call(key, RelLock, obj)$$

$$LockReleased(key, obj) \equiv$$
$$Return(key, RelLock, obj)$$
$$\wedge\ \neg locked(key, obj)'$$

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 *dependency* 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) $Initial \Rightarrow A$; and (ii) $A \wedge B \wedge e \Rightarrow A'$ for every event e [4]. *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 *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:

$B_1 \equiv \neg allocated(key) \Rightarrow (\forall id : keyof(id) \neq key)$

$B_2 \equiv allocated(key) \Rightarrow (\exists \text{exactly one } id : keyof(id) = key)$

Lemma 1. $B_1 \wedge B_2$ is invariant, given *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:

$C_1 \equiv status_U(id) \in \{\text{NOTBEGUN}, (BeginTr)\} \Rightarrow (id) \notin H_U$

$C_2 \equiv keyof(id) = key \wedge status_U(id) = \text{READY} \Rightarrow status_L(key) = \text{READY}$

$C_3 \equiv keyof(id) = key \wedge \neg locked(key, obj) \Rightarrow (id, obj) \notin H_U$

The consequent of C_3 implies $currentvalue(obj, key) = committedvalue(obj)$.

$C_4 \equiv keyof(id) = key \wedge status_L(key) = (AcqLock, obj)$
$\Rightarrow \neg locked(key, obj) \wedge status_U(id) = (key, obj)$

$C_5 \equiv keyof(id) = key \wedge locked(key, obj) \wedge status_U(id) \neq (EndTr)$
$\Rightarrow storedvalue(obj) = committedvalue(obj)$

$C_6 \equiv keyof(id) = key \wedge status_U(id) = (key, obj) \wedge locked(key, obj)$
$\wedge\ localvalue(obj, key) = \text{NULL} \Rightarrow (id, obj) \notin H_U$

$C_7 \equiv keyof(id) = key \wedge status_L(key) = (Read, obj) \Rightarrow locked(key, obj)$
$\wedge\ localvalue(obj, key) = \text{NULL} \wedge (id, obj) \notin H_U \wedge status_U(id) = (ReadTr, key, obj)$

$C_8 \equiv keyof(id) = key \wedge locked(key, obj) \Rightarrow obj \in accessed(id)$

$A_1 \equiv keyof(id) = key \wedge localvalue(obj, key) \neq \text{NULL}$
$\Rightarrow localvalue(obj, key) = currentvalue(obj, id)$

where we have repeated A_1 for convenience of reference.

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

$D_1 \equiv keyof(id) = key \wedge locked(key, obj) \wedge status_U(id) = (EndTr) \wedge localvalue(obj, key) \neq \text{NULL}$
$\Rightarrow storedvalue(obj) = committedvalue(obj)$

$D_2 \equiv keyof(id) = key \wedge status_L(key) = (Write, obj, val)$
$\Rightarrow status_U(id) = (EndTr, key) \wedge val = currentvalue(obj, id) \wedge locked(key, obj)$

$D_3 \equiv keyof(id) = key \wedge locked(key, obj) \wedge status_U(id) = (EndTr) \wedge localvalue(obj, key) = \text{NULL}$
$\Rightarrow storedvalue(obj) = currentvalue(obj, id)$

The following assertions specify relationships during the lock releasing phase of a transaction:

$E_1 \equiv \neg allocated(key) \Rightarrow localvalue(obj, key) = \text{NULL}$

$E_2 \equiv status_L(key) = (RelLock, obj) \Rightarrow \neg allocated(key) \wedge locked(key, obj)$

$E_3 \equiv \neg locked(key, obj) \Rightarrow localvalue(obj, key) = \text{NULL}$

Two additional assertions are needed:

$F_1 \equiv (\forall key : \neg locked(key, obj)) \Rightarrow storedvalue(obj) = committedvalue(obj)$

$F_2 \equiv owned(key, obj) \Leftrightarrow locked(key, obj)$

We use the notation $C_{1\text{-}8}$ to denote the conjunction $C_1 \wedge C_2 \cdots \wedge C_8$.

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

Lemma 2 establishes A_1. We next show that it also establishes A_2. Assume *deadlock*(key, obj) to be true. From the definition of *deadlock*, we have *waiting*(key, obj) and *owned*(k, obj) for some $k \neq key$. From the definition of *waiting* and C_4, *waiting*(key, obj) implies $status_U(id) = (obj)$ for some *id*, which implies $obj \in accessed(id)$. From F_2 and C_8, *owned*(k, obj) implies $obj \in accessed(id_1)$ for some id_1. From $B_1 \wedge B_2$, we have $id_1 \neq id$. Consequently, *concurrentaccess*(id) is true.

5.2. Proof of serializability

The following assertions are to be proved:

$$G_1 \equiv (id_1, obj) \in H_U \wedge (id_1, EndTr) \notin H_U \wedge (id_1, AbortTr) \notin H_U \Rightarrow keyof(id_1) \neq \text{NULL} \wedge locked(keyof(id_1), obj)$$

$$G_2 \equiv (id_1, obj) @ (id_2, obj) \text{ is a subsequence of } H_U \Rightarrow (id_1, obj) @ (id_1, EndTr) @ (id_2, obj) \text{ is a subsequence of } H_U \vee (id_1, obj) @ (id_1, AbortTr) @ (id_2, obj) \text{ is a subsequence of } H_U$$

Lemma 3. $G_1 \wedge G_2$ is invariant, given $B_1 \wedge B_2 \wedge legal(id_1)$. (Proof omitted.)

Lemma 3 implies that the following is invariant:

$$G_3 \equiv dependency(id_1, id_2) \Rightarrow (id_1, EndTr) @ (id_2, EndTr) \text{ is a subsequence of } C(H_U)$$

This can be proved as follows. Given *dependency*(id_1, id_2). From the definitions of $C(H_U)$ and *legal*(id_2), $(id_2, obj) \in C(H_U)$ implies that $(id_2, obj) @ (id_2, EndTr)$ is a subsequence of $C(H_U)$. Combining this with G_2, we see that $(id_1, obj) @ (id_2, obj)$ is a subsequence of $C(H_U)$ implies $(id_1, obj) @ (id_1, EndTr) @ (id_2, obj) @ (id_2, EndTr)$ is a subsequence of $C(H_U)$.

G_3 implies that *dependency* is acyclic. Assume the contrary: For some $n \geq 2$, there exist distinct $id_1, id_2, \cdots, id_n$, such that *dependency*(id_k, id_{k+1}) for $k = 1, \cdots, n-1$, and *dependency*(id_n, id_1). From G_3, $C(H_U)$ contains the subsequences $(id_k, EndTr) @ (id_{k+1}, EndTr)$, for $k = 1, \cdots, n-1$, and $(id_n, EndTr) @ (id_1, EndTr)$. But this implies that there are at least two occurrences of $(id_1, EndTr)$ in $C(H_U)$, which violates *legal*(id_1).

5.3. Proof of progress

Given predicates A and B and an event e_1, we say that A *leads-to* B *via* e_1 if the following are logically valid: (i) $A \Rightarrow enabled(e_1)$, (ii) $A \wedge e_1 \Rightarrow B'$, and (iii) $A \wedge e \Rightarrow A' \vee B'$ for every event e [4]. Whenever A holds, parts (i) and (ii) ensure that e_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 *leads-to* to denote the closure of the *leads-to-via* relation; e.g., A *leads-to* B if A *leads-to* $B \vee C$ and C *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 *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 *leads-to* B, we need to establish that $A \Rightarrow b$ is invariant.

Consider Q_1, which is achieved by executing the *Return* (*key* ,*Read* ,*obj* ,*val*) event. This event is imbedded in the implementation event *ReadCompleted* (*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 *WriteCompleted* (*key* ,*obj*) is enabled whenever *Return* (*key* , *Write* ,*obj* ,*val*) is enabled.

We can assume Q_3 because *LockReleased* (*key* ,*obj*) is enabled whenever *Return* (*key* ,*RelLock* ,*obj*) is enabled.

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

From Q_3, *ReqRelLock* , and *LockReleased* , we can establish:

$$holdinglocks(key) \wedge \neg allocated(key) \ \ leads\text{-}to \ \ \neg holdinglocks(key)$$

From this and *Return* (*BeginTr*), we can establish:

$$W_1 \equiv status_U(id)=(BeginTr) \ \ leads\text{-}to \ \ status_U(id) \in \{\text{READY,NOTBEGUN}\}$$

From Q_2, *ReqWrite* , and *WriteCompleted* , we have:

$$status_U(id)=(EndTr,key) \wedge localvalue(obj,key) \neq \text{NULL}$$
$$leads\text{-}to \ \ status_U(id)=(EndTr,key) \wedge localvalue(obj,key)=\text{NULL}$$

From *Return* (*EndTr* ,*key*), we have:

$$status_U(id)=(EndTr,key) \wedge localvalue(obj,key)=\text{NULL}$$
$$leads\text{-}to \ \ status_U(id)=\text{COMMITTED}$$

Combining the above two, we have:

$$W_2 \equiv status_U(id)=(EndTr,key) \ \ leads\text{-}to \ \ status_U(id)=\text{COMMITTED}$$

From *Return* (*AbortTr*), we have:

$$W_3 \equiv status_U(id)=(AbortTr) \ \ leads\text{-}to \ \ status_U(id)=\text{ABORTED}$$

From Q_1, *ReqRead* and *ReadCompleted* , we have:

$$status_U(id)=(ReadTr,key,obj) \wedge locked(key,obj)$$
$$leads\text{-}to \ \ status_U(id)=(ReadTr,key,obj) \wedge localvalue(obj,key) \neq \text{NULL}$$

From above and *Return* (*ReadTr* ,*val*), we have:

$$W_4 \equiv status_U(id)=(ReadTr,key,obj) \wedge locked(key,obj) \ \ leads\text{-}to \ \ status_U(id)=\text{READY}$$

From *Return* (*WriteTr* ,OK), we have:

$$W_5 \equiv status_U(id)=(WriteTr,key,obj) \wedge locked(key,obj) \ \ leads\text{-}to \ \ status_U(id)=\text{READY}$$

From *RequestLock*, we have:

$W_6 \equiv status_U(id)=(key,obj) \wedge \neg locked(key,obj)$ *leads-to* $waiting(key,obj)$

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:

$$status_U(id)=(key,obj) \wedge waiting(key,obj) \textit{ leads-to } \neg waiting(key,obj).$$

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

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

Define the following state functions on the directed graph, where $1 \le j \le M$:

$waitstate_1(j)$: boolean

True iff there are $k_2, x_2, \cdots, k_j$ such that $k_1, x_1, \cdots, k_j$ is a path and k_j is not waiting.

$waitstate_2(j)$: boolean

True iff there are $k_2, x_2, \cdots, k_j, x_j$ such that $k_1, x_2, \cdots, k_j$ is a path, and $waiting(k_j,x_j)$ and x_j is not locked.

$waitstate_3(j)$: boolean

True iff there are $k_2, x_2, \cdots, k_j$ such that $k_1, x_2, \cdots, k_j$ is a path, and $waiting(k_j,x_i)$ for some x_i such that $1 \le i < j$ or $owned(k_1,x_i)$; i.e., k_j is deadlocked.

Observe that for any state of the directed graph, exactly one of the $3M$ functions $\{waitstate_1(j), waitstate_2(j), waitstate_3(j): 1 \le j \le M\}$ is true. At any time, let the function *depth* denote that value of j.

Define the following functions, where $1 \le i \le M$:

$\beta(i)$: integer

If $i < depth$, or if $i = depth$ and k_i is waiting: $\beta(i)$ equals the number of times x_i has been unlocked since the last time that k_i started to wait.

If $i = depth$ and k_i is not waiting: $\beta(i)=0$.

If $i > depth$: $\beta(i)=-1$.

$\alpha(i)$: integer

If $i < depth$, or if $i = depth$ and k_i is not releasing locks (i.e., $allocated(k_i)$ is true): $\alpha(i)$ equals the number of objects held by k_i.

If $i = depth$ and k_i is releasing locks: $\alpha(i)$ equals the number of objects locked by k_i just before it started releasing locks.

If $i > depth$: $\alpha(i)=0$.

Define the function $\alpha = (\beta(1), \alpha(2), \beta(2), \alpha(3), \cdots, \alpha(M), \beta(M))$. The values of α can be well-ordered using the lexicographic ordering. We will show that α increases without bound unless $\neg waiting(k_1,x_1)$ becomes true. This establishes the desired progress property as follows: α

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

The following *leads-to* properties can be established.

$W_7 \equiv$ $waitstate_1(j) \wedge \alpha=\mathbf{a}$ *leads-to* $W_{7a} \vee W_{7b} \vee W_{7c} \vee W_{7d}$, where

$W_{7a} \equiv waitstate_2(j-1) \wedge \alpha=\mathbf{a}[\beta(j-1)=b_{j-1}+1;\ \alpha(j)=0;\ \beta(j)=-1] > \mathbf{a}$

$W_{7b} \equiv waitstate_1(k) \wedge j<k \ \wedge \alpha=\mathbf{a}[\beta(i)=0;\ \alpha(i)\geq 0\text{: } j<i\leq k] > \mathbf{a}$

$W_{7c} \equiv waitstate_2(k) \wedge j\leq k \ \wedge \alpha=\mathbf{a}[\beta(i)=0;\ \alpha(i)\geq 0\text{: } j<i\leq k] \geq \mathbf{a}$

$W_{7d} \equiv waitstate_3(k) \wedge j\leq k \ \wedge \alpha=\mathbf{a}[\beta(i)=0;\ \alpha(i)\geq 0\text{: } j<i\leq k] \geq \mathbf{a}$

W_{7a} results if k_j returns the lock on x_{j-1}, in the process of releasing its locks. The value of α 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. $W_{7b} \vee W_{7c} \wedge j<k$ 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 $W_{7b} \vee W_{7c} \vee W_{7d}$, if $k>j$, the value of α 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 $k=j$, α stays constant. One of the above transitions will eventually occur because k_j is ready in $waitstate_1(j)$, i.e., $status_U(id)$=READY where $keyof(id)=k_j$.

$W_8 \equiv$ $waitstate_2(j) \wedge \alpha=\mathbf{a}$ *leads-to* $\neg waiting(k_1,x_1) \vee W_{8a} \vee W_{8b}$, where

$W_{8a} \equiv waitstate_1(j) \wedge \alpha=\mathbf{a}[\alpha(j)=a_j+1;\ \beta(j)=0] > \mathbf{a}$

$W_{8b} \equiv waitstate_1(j+1) \wedge \alpha=\mathbf{a}[\beta(j+1)=0;\ \alpha(j+1)\geq 0] > \mathbf{a}$

The $LockAcquired(k_j,x_j)$ event is continuously enabled in $waitstate_2(j)$. Its occurrence results in $\neg waiting(k_1,x_1)$ if $j=1$, and in W_{8a} if $j>1$. This will eventually occur unless x_j is locked by a key other that k_j. In this case, that key becomes k_{j+1} and W_{8b} holds. In the case of W_{8a}, the value of α increases because $\alpha(j)$ increases and it is the most significant function changed. In the case of W_{8b}, the value of α increases because $\beta(j+1)$ increases from -1 to 0, $\alpha(j+1)$ stays at 0 or increases, and no other functions change values.

$W_9 \equiv$ $waitstate_3(j) \wedge \alpha=\mathbf{a}$ *leads-to* $\neg waiting(k_1,x_1) \vee W_{9a}$, where

$W_{9a} \equiv waitstate_2(k) \wedge j>k \ \wedge \alpha=\mathbf{a}[\beta(k)=b_k+1;\ \beta(i)=0;\ \alpha(i)=-1\text{: } k<i\leq j] > \mathbf{a}$

$waitstate_3(j)$ implies a cycle involving k_j. This cycle only involves keys from $k_1, k_2, \cdots, k_j$. $LockRejected(k_i,x_i)$ for every k_i involved in the cycle is enabled, and the lock manager will execute one of them eventually. $\neg waiting(k_1,x_1)$ results if k_1 is involved in the deadlock and $LockRejected(k_1,x_1)$ 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 α 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 $waitstate_1(j) \wedge \alpha=\mathbf{a}$ *leads-to* $\alpha>\mathbf{a} \vee \neg waiting(k_1,x_1)$. From W_8, we have $waitstate_2(j) \wedge \alpha=\mathbf{a}$ *leads-to* $\alpha>\mathbf{a}$ $\vee \neg waiting(k_1,x_1)$. From W_9, we have $waitstate_3(j) \wedge \alpha=\mathbf{a}$ *leads-to* $\alpha>\mathbf{a}$

$\vee \neg waiting(k_1,x_1)$. Combining these three *leads-to* statements, we have $\alpha=\mathbf{a}$ *leads-to* $\alpha>\mathbf{a} \vee \neg waiting(k_1,x_1)$, which establishes that α 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

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Massachusetts, 1987.

[2] Simon S. Lam and A. Udaya Shankar, "Protocol Verification via Projections," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 10, July 1984, pp. 325-342.

[3] A. Udaya Shankar and Simon S. Lam, "An HDLC Protocol Specification and Its Verification Using Image Protocols," *ACM Transactions on Computer Systems*, Vol. 1, No. 4, November 1983, pp. 331-368.

[4] A. Udaya Shankar and Simon S. Lam, "Time-dependent distributed systems: proving safety, liveness and real-time properties," *Distributed Computing*, Vol. 2, No. 2, 1987; available as technical report UMIACS-TR-85-4.1, University of Maryland, College Park, and as technical report TR-85-24, University of Texas at Austin.

[5] A. Udaya Shankar and Simon S. Lam, "A Stepwise Refinement Heuristic for Protocol Construction," technical report UMIACS-TR-87-12, University of Maryland, College Park, March 1987.