SPECIFYING IMPLEMENTATIONS TO SATISFY INTERFACES: A STATE TRANSITION SYSTEM APPROACH[*]

Simon S. Lam and A. Udaya Shankar[†]

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

TR-88-30 August 1988
June 1989 (Revision)

[*]The work of Simon S. Lam was supported by National Science Foundation Grant no. NCR-8613338. The work of A. Udaya Shankar was supported by National Science Foundation Grant no. ECS-8502113.

[†]Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742.

Specifying Implementations to Satisfy Interfaces: A State Transition System Approach*

Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

A. Udaya Shankar
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

August 1988

Second revision, June 1989

Abstract

We present a formalism to specify interfaces and implementations of program modules in a hierarchy. Our formalism is based upon the relational notation for specifying state transition systems and a refinement relation between such specifications. We define what it means for a program module to *offer an upper interface* to a user, and to *use a lower interface* offered by another program module. We then solve the problem posed by Leslie Lamport to participants of the Specification Logics session of the 1987 Lake Arrowhead Workshop. A formal specification of a serializable database interface is first presented. Specifications of two database implementations, using a two-phase locking protocol and a multi-version timestamp protocol, are then given, together with a proof that each implementation satisfies the interface. In the two-phase locking implementation, we assume that it uses a lower interface to access a physical database.

*The work of Simon S. Lam was supported by National Science Foundation Grant no. NCR-8613338. The work of A. Udaya Shankar was supported by National Science Foundation Grant no. ECS-8502113.

1. Introduction

We consider the specification of program modules that interact via interfaces. The program modules, or simply *modules*, are organized in a hierarchy. Each module in the hierarchy *offers an upper interface* to a user, which may be a module at a higher level of the hierarchy. Each module may *use a lower interface* offered by a module at a lower level of the hierarchy. (See Figure 1.) While the upper interface of a module can be offered to at most one user, the module can make use of multiple lower interfaces each offered by a different module at some lower level of the hierarchy. Thus, the hierarchy is, in general, a tree with tree nodes being modules; each tree node offers an upper interface to its father while making use of lower interfaces to access its sons.

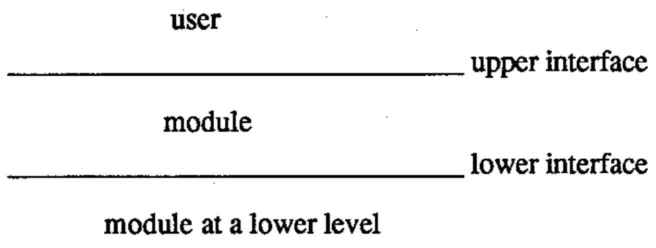

Figure 1. A module and its environment.

An interface between a module and its user can be characterized by a set of *input events*, a set of *output events*, and a set of *allowed* sequences of the input and output events. Each allowed event sequence represents one possible interaction between the user and the module. Input events are controlled by the user. Output events are controlled by the module. This view is similar to that of Lynch and Tuttle [13],

While it is conceptually simple to view an interface as a set of allowed event sequences, some convenient way of specifying this set is needed in practice. In Section 2, we present one such specification formalism. Before doing so, we discuss informally what it means for a module that offers an upper interface and uses a lower interface, to satisfy its interfaces. (A more formal treatment is given in Section 2.)

In our formalism, an interface is specified by a state transition system, a set of invariant requirements, and a set of progress requirements. Each event of the state transition system is either an input event or an output event of the interface.

We specify a module by a state transition system and a set of fairness requirements for some of its events. Suppose the module offers an upper interface and uses a lower interface, as illustrated in Figure 1. By design, some module events are said to *correspond* to certain events of its upper and lower interfaces. (This notion of event correspondence will be made precise in Section 2.5.) We require that each module event corresponds to at most one interface event. The correspondence between module and interface events induces a correspondence between event sequences. For any sequence u of module events, the image of u on the upper (lower) interface is defined to be the sequence obtained from u by deleting from it, module events that do not correspond to any upper (lower) interface event, and replacing the remaining module events in it by their corresponding interface events.

We say that the module satisfies its interfaces if and only if (iff) every event sequence u allowed by the module satisfies this condition: if the image of u on the lower interface is an allowed sequence of the lower interface, then the image of u on the upper interface is an allowed sequence of the upper interface. In our formalism, presented in Section 2, the above condition is met if

- the module specification is a refinement [8] of each of the interface specifications, and
- it satisfies the invariant and progress requirements of the upper interface assuming that the invariant and progress requirements of the lower interface are satisfied.

The above definition of what it means for a module to satisfy its interfaces extends in a straightforward manner to modules that offer an upper interface while using multiple lower interfaces.

Our basic notion of a module satisfying an offered interface is essentially the same as that in other state-transition formalisms [4,10,11,13]. The working definition in our formalism is based upon the refinement relation between state transition systems in [8], which is adapted from our earlier work on projection mappings between state transition systems [6,21]. Our formal treatment of the more general notion of offering an upper interface while using some lower interfaces appears to be new.

Other authors allow arbitrary compositions of modules [5,13]. Our approach is not so general. But by imposing a hierarchical relationship between modules that interact via an interface, we get fairly simple conditions for composing the modules. In our experience, we have not found the hierarchical structure of modules assumed here to be restrictive. This is because a module in our formalism can be an arbitrary network of processes. In Section 7, we give a more detailed comparison of our method with other approaches in the literature, and describe various other applications of our method.

1.1. Database examples

In Figure 2, the problem posed by Lamport is illustrated. The database system is a module. The client programs together constitute the user of the module. It is assumed that client programs execute concurrently. Each issues a sequence of transactions to be processed by the database system. We refer to the interface offered by the database system to the client programs as the database interface. We will provide formal specifications of the database system and the database interface. However, the module representing the client programs will not be explicitly specified.

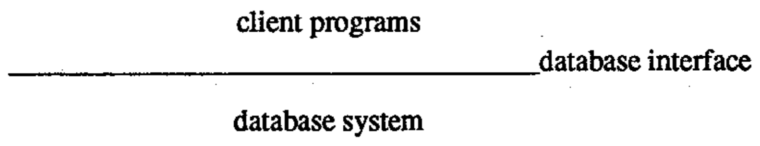

Figure 2. A database system.

Lamport's informal specification of an interface consists of a set of procedures that can be executed concurrently [12]. We model such an interface procedure P by two events: $Call(P)$ and $Return(P)$. Since several invocations of P can be concurrently active, we tag each call of P with a unique identifier, which is also 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 is unique over all concurrent invocations of P. A call event is an input event of the interface. A return event is an output event of the interface.

In specifying the database system module, each procedure is also modeled by two events *Call* (i, P) and *Return* (i, P), which are obtained by refining the corresponding events of the interface. Because the action of each event in our formalism is atomic, the atomic actions in our module specifications may be too large — in the following sense. For a practical programming language, such as Pascal or C, a procedure execution consists of a call event occurrence, followed by occurrences of events that constitute the procedure body, and concluded by a return event occurrence. State variables are updated by events in the procedure body. The call and return events can only transfer control and parameter values. Thus for implementation in a practical programming language, the module specifications given in this paper will have to be refined further; specifically, state variables that are updated in the actions of *Return* (i, P) events will have to be made "auxiliary variables." In Section 6, we indicate how such refinements can be carried out.

1.2. The balance of this paper

In Section 2, we present our formalism for specifying state transition systems, safety and progress properties, interfaces, and modules. The refinement relation between state transition systems is introduced. Sufficient conditions for a module to satisfy its interfaces are formally stated.

In Section 3, the database interface is specified. Two implementations of the database system are then specified, one using a two-phase locking protocol in Section 4, and the other using a multi-verison timestamp protocol in Section 5. The two-phase locking implementation, as specified informally by Lamport, makes use of a lower interface to access a physical database; we formally specify this interface also in Section 4. For each implementation, we provide a proof that it satisfies the database interface. (Our progress proof for the two-phase locking implementation employs a novel metric based upon lexicographic ordering [7].)

In Section 6, we discuss how to refine *Return* (i, P) events to satisfy the atomicity requirements of a practical programming language. In Section 7, we discuss related work by other authors and compare our approach with those of others.

2. Model and Proof Method

We first present the relational notation for specifying state transition systems introduced in [8, 22].

2.1. State transition systems

A state transition system is specified by a set of *state variables*, $\mathbf{v} = \{v_1, v_2, \cdots\}$, a set of *events*, $e_1, e_2, \cdots$, and an initial condition, as defined below. For every state variable, there is a specified domain of allowed values. The system state is represented by the set of values assumed by the state variables. Parameters may be used for defining groups of related events, as well as groups of related system properties. Let $\mathbf{w}$ denote a set of parameters, each with a specified domain of allowed values.

Let $\mathbf{v}'$ denote the set of variables $\{v' \colon v \in \mathbf{v}\}$. In specifying an event, we use $\mathbf{v}$ and $\mathbf{v}'$ to denote, respectively, the system state before and after an event occurrence. Instead of a programming language, the language of predicate logic is used for specifying events. We assume that there is a known interpretation that assigns meanings to all of the function symbols and predicate symbols, and values to all of the constant symbols that we use. As a result, the truth value of a formula can be determined if values are assigned to its free variables.[1]

[1]We use *formula* to denote *well-formed formula*.

The set of variables in our language is $\mathbf{v} \cup \mathbf{v}' \cup \mathbf{w}$. We will use two kinds of formulas: A formula whose free variables are in $\mathbf{v} \cup \mathbf{w}$ is called a *state formula.* A formula whose free variables are in $\mathbf{v} \cup \mathbf{v}' \cup \mathbf{w}$ is called an *event formula.*

A state formula can be evaluated for each system state to be true or false by adopting this convention: if a parameter occurs free in a state formula, it is assumed to be universally quantified over the parameter domain.

We say that a system state s satisfies a state formula F iff F evaluates to true for s. A state formula F represents the *set* of system states that satisfy F. In particular, the initial condition of the system is specified by a state formula. A system state that satisfies the initial condition is called an *initial state.*

Events are specified by event formulas. Each event (formula) defines a *set* of system state transitions. Some examples of event definitions are shown below:

$$e_1 \equiv v_1>2 \wedge v_2' \in \{1,2,5\}$$

$$e_2 \equiv v_1>v_2 \wedge v_1+v_2'=5$$

where " $\equiv$ " denotes "is defined by"; in each definition, the event name is given on the left-hand side and the event formula is given on the right-hand side. For convenience, we sometimes use the same symbol to denote the name of an event as well as the event formula that defines it. The context where the symbol appears will determine what it means.

Convention. Given an event formula e, for every state variable v in $\mathbf{v}$, if v' is not a free variable of e then each occurrence of the event e does not change the value of v; that is, the conjunct $v'=v$ is implicit in the event formula.

For example, consider a system with two state variables v_1 and v_2. Let e_2 above be an event of the system. The conjunct $v_1'=v_1$ is implicit in the above formula that defines e_2.

If a parameter occurs free in an event definition, then the system has an event defined for each value in the parameter domain. For example, consider

$$e_3(m) \equiv v_1>v_2 \wedge v_1+v_2'=m$$

where m is a parameter. A parameterized event is a convenient way to specify a group of related events.

An event can occur only when the system state satisfies the enabling condition of the event. In any system state, more than one event may be enabled. The choice of the next event to occur from the set of enabled events is nondeterministic.[2] When an event occurs, we assume that the state variables of the system are updated in one *atomic* step.

Formally, the enabling condition of an event formula e, to be denoted by $enabled(e)$, is given by

$$enabled(e) \equiv [\exists \mathbf{v}' \colon e]$$

which is a state formula.

Some of the state variables in $\mathbf{v}$ may be *auxiliary variables,* which are needed for specification or verification only, and do not have to be included in an actual implementation of the specification.[3] For example, an auxiliary variable may be needed to record the history of certain event occurrences.

[2]The choice is not strictly nondeterministic if the system specification includes fairness requirements for some events.

[3]A rigorous explanation of this statement can be found in [8]. What we call auxiliary variables here are also known as history variables. Abadi

Informally, a subset of variables in **v** is auxiliary if they do not affect the enabling condition of any event nor do they affect the update of any state variable that is not auxiliary. To state the above condition precisely, let **u** be a proper subset of **v**, and $\mathbf{u}' = \{v' : v \in \mathbf{u}\}$. The state variables in **u** are auxiliary if, for every event e of the system, the following holds:

$$e \Rightarrow [\forall \mathbf{u} \exists \mathbf{u}' : e]$$

For convenience in specification, we will also use *state functions*, that is, functions of the system state. For example, we can define a boolean state function *even* such that $even(v)$ is true iff the value of the state variable v is an even integer. (Note that state functions can always be transformed into state variables.)

Each execution of a state transition system can be represented by a *behavior*, namely, a sequence $<s_0, f_0, s_1, f_1, \cdots>$ of alternating states and events, where s_0 is an initial state, and for each i, s_i denotes the system state at the start of the ith transition, f_i denotes an event that is enabled in s_i, and the transition (s_i, s_{i+1}) satisfies f_i. A behavior may be finite or infinite. By definition, a finite behavior ends in a state. Safety properties of the system are determined by the set of its finite behaviors.

For a system to have desirable progress properties, it may be necessary that some events must be scheduled to satisfy a fairness criterion. Note that a state transition system in itself does not have any notion of fairness. Therefore, in specifying a state transition system using the relational notation, we explicitly include a set of fairness requirements. The fairness requirements should be as weak as possible to facilitate implementation. Generally, to achieve certain progress properties, only some of the events in a specification need to be fairly scheduled. Very often, we define a new event to be the disjunction of a set of events already defined, e.g.,

$$e_3 \equiv [\exists m : e_3(m)]$$

$$e_4 \equiv e_1 \vee e_2$$

because it is the new event that needs to be fairly scheduled, and not the individual events in the set.

The only fairness criterion we will use in this paper is *weak fairness*, also called *justice* [15]. Informally, if an event e having weak fairness is continuously enabled, it eventually occurs. More precisely, consider a specification consisting of a state transition system and some fairness assumptions. An infinite behavior w satisfies an assumption of weak fairness for event e iff e occurs infinitely often or is disabled infinitely often in w.

The *allowed behaviors* of the specification are defined as follows: [4]

- A finite behavior of the state transition system is an allowed behavior of the specification iff every event that has a fairness assumption in the specification is disabled in the last state of the behavior.
- An infinite behavior of the state transition system is an allowed behavior of the specification iff every fairness assumption of the specification is satisfied by the behavior.

The progress properties of the specification are determined by the set of its allowed behaviors.

2.2. Safety and progress properties

We use two kinds of assertions to describe the behaviors allowed by a specification. For safety properties, we use assertions of the form: P is invariant, where P is a state formula. P is invariant for a

and Lamport [1] defined another kind of auxiliary variables called prophecy variables.

[4]This definition is applicable for specifications in which each event may be scheduled with no fairness, weak fairness or strong fairness; see [8]. What we define to be an allowed behavior is called a fair behavior by Lynch and Tuttle [13].

behavior w iff every state of w satisfies P. P is invariant for a state transition system iff P is invariant for every finite behavior of the system.

To state proof rules for assertions, we need the following notation: for an arbitrary state formula R, R' denotes the formula obtained from R by replacing every state variable v in it by v'.

For any state transition system A, let $\mathbf{v}_A$ denote its set of state variables, and $Initial_A$ be a state formula specifying its initial condition.

Invariance rule: P is invariant for state transition system A if

- $Initial_A \Rightarrow P$, and
- for every event e of A, $P \wedge e \Rightarrow P'$

If I is invariant for A, then we can replace $P \wedge e \Rightarrow P'$ with $I \wedge I' \wedge P \wedge e \Rightarrow P'$ in the above rule. If P is invariant for a system, then any state formula implied by P is also invariant for the system.

For progress properties, we use assertions of the form: P *leads−to* Q, where P and Q are state formulas. P *leads−to* Q for a behavior w iff the following holds: if some state s_i in w satisfies P then there is a state s_j in w, $j \geq i$, that satisfies Q. P *leads−to* Q for a specification iff P *leads−to* Q for every allowed behavior of the specification. Some rules for proving leads-to assertions are given below.[5]

Leads-to rules: For a given specification, P *leads−to* Q if one of the following holds:

- $P \Rightarrow Q$ is invariant [implication]
- for some event e that has fairness, P *leads−to* Q *via* e [event]
- for some R, P *leads−to* R and R *leads−to* Q [transitivity]
- $P = P_1 \vee P_2$, P_1 *leads−to* Q and P_2 *leads−to* Q [disjunction]

where the following definition is assumed:

Definition: For a given specification in which event e has weak fairness, P *leads−to* Q *via* e iff

(i) $P \wedge e \Rightarrow Q'$,

(ii) for every event f, $P \wedge f \Rightarrow P' \vee Q'$, and

(iii) $P \Rightarrow enabled(e)$ is invariant.

If I is invariant for the specification, it can be used to strengthen the antecedent of each logical implication in the above definition; that is, replace P by $I \wedge I' \wedge P$ in each part of the definition. For a given specification, if I is invariant and $P \wedge I$ *leads−to* Q, then we infer that P *leads−to* Q.

2.3. Refinement of state transition systems

We next define a refinement relation between two state transition systems A and B. Let the state variable set of system A be $\mathbf{v}_A = \{v_1, v_2, \cdots, v_n\}$ and the state variable set of system B be $\mathbf{v}_B = \{v_1, v_2, \cdots, v_m\}$, where $m \leq n$. That is, in deriving A from B, every state variable in B is kept as a state variable in A with the same name and the same domain of values. Since $\mathbf{v}_B$ is a subset of $\mathbf{v}_A$,

[5] For a comprehensive treatment of proof rules, the reader is referred to [4,15,19]. For distributed systems with unreliable communication channels, see [8] for the *P leads-to Q via M* rule, where *M* denotes a set of messages.

there is a projection mapping from the states of A to the states of B, defined as follows: The set of states in A having the same values for $\{v_1, v_2, \cdots, v_m\}$ are mapped to the same state in B [6,8,21]. We further require that every parameter in B is a parameter in A with the same name and same domain of values. Given the above requirements, any state formula of system B can be interpreted directly for system A without translation.

Let $\{a_i\}$ denote the set of events of system A, and $\{b_j\}$ the set of events of system B. Event a_i in system A is a refinement of events in system B if, for some invariant I_A of system A, one of the following holds:

$$I_A \wedge a_i \Rightarrow [\exists j: b_j] \qquad \text{(event refinement condition)}$$

$$I_A \wedge a_i \Rightarrow v_1'=v_1 \wedge v_2'=v_2 \wedge \cdots \wedge v_m'=v_m \qquad \text{(null image condition)}$$

Very often, a_i is the refinement of a single event in system B. In this case, to check that a_i satisfies the refinement condition, it is sufficient to show that, for some b_j, either $a_i \Rightarrow b_j$ or $I_A \wedge a_i \Rightarrow b_j$.

It might appear that the use of a multi-valued possibilities mapping [13] between A and B is more general than the use of a projection mapping, as described above. This is not the case, however, because we allow state variables in B to be replaced by new state variables in A, and then made into auxiliary variables in A. To prove that events of A and events of B satisfy the event refinement and null image conditions, an invariant I_A is to be found. This invariant represents a multi-valued possibilities mapping from the values of the old state variables to the values of the new state variables.

System A is a *refinement* of system B iff every event in A is a refinement of events in B and $Initial_A \Rightarrow Initial_B$. This ensures that any invariant property of B is an invariant property of A. (By imposing additional conditions, we can ensure that some, or all, leads-to properties of B are also leads-to properties of A [8].)

Suppose A is a refinement of B. For this paper, A represents a module implementation and B a module interface. We impose the additional requirement that every event of A that does not have a null image, is a refinement of a *single* event of B; that is, every event of A has a unique image in B. For any sequence $w=<s_0, a_0, s_1, \cdots>$ of alternating states and events of A, we define the image of w in B to be the sequence obtained from w by (i) removing every (a_i, s_{i+1}) pair in w such that a_i has a null image, and (ii) replacing every remaining state and event in w by their images in B. The requirement that every event of A has a unique image (which may be null) guarantees that the image of any behavior of A is a behavior of B.

2.4. Specifying interfaces

The specification of a module interface B consists of the following:

- A state transition system B, whose events are divided into input events and output events.
- A set I_B of invariant requirements.
- A set L_B of progress requirements.

In specifying an interface, we do not need to know whether it is the upper or lower interface of a module. For a given specification, a behavior of the state transition system that also satisfies the invariant and progress requirements is said to be an *allowed behavior of the interface.* An event sequence is *allowed* by the interface iff it is the sequence of events in an allowed behavior of the interface. (Note that an allowed behavior may be finite or infinite.)

Note that I_B is a set of state formulas that are required to be invariant for B. For notational convenience, we will also use I_B to denote a state formula that is the conjunction of all the state formulas in the set I_B; I_B' is a formula obtained from the state formula I_B by replacing every state variable v in it by v'.

For every event f of the interface, define

$$possible(f) \equiv I_B \wedge [\exists v_B': f \wedge I_B']$$

The formula *possible*(*f*) is a state formula. It is true for every interface state where the event f is allowed to occur. Note that $possible(f) \Rightarrow enabled(f)$ holds.

In the above, we have provided two ways to specify the safety requirements of an interface: namely, a state transition system, and a set of invariant requirements. It is our experience that some safety requirements are more easily expressed by invariant requirements, while some are more easily expressed by state variables and events. Our approach is a flexible one. We could use a state transition system that is very small. In the extreme case, a single state variable is enough, namely, a trace variable recording the sequence of all event occurrences. Each event is always enabled and its action consists of only updating the trace variable. On the other hand, we could try to specify safety requirements using a state transition system only, without any invariant requirement; however, doing so generally results in a cumbersome state transition system.

2.5. Specifying modules

Consider a module A that offers an *upper* interface B and uses a *lower* interface C. Interface B (C) is specified by a state transition system B (C), a set of invariant requirements I_B (I_C) and a set of progress requirements L_B (L_C). An implementation of the module is specified by a state transition system A together with a set of weak fairness requirements for events.

As defined in Section 2.1, a behavior w of the state transition system A is an *allowed behavior of the module* if and only if

- w is infinite and w satisfies every fairness requirement of the module
- w is finite and no event with a fairness requirement is enabled in the last state of w.

An event sequence is *allowed* by the module iff it is the sequence of events in an allowed behavior of the module. As defined in the Introduction, module A satisfies its interfaces iff every event sequence u allowed by A satisfies this condition: if the image of u on the lower interface C is an allowed sequence of C, then the image of u on the upper interface B is an allowed sequence of B.

We present here a set of sufficient conditions for a module to satisfy its interfaces. These will be referred to as **module implementation conditions:**

M1. A is a refinement of C. Additionally, the only way for an event of A to access state variables of C is by including an event of C as a conjunct in its definition. We say that module event e *corresponds* to lower interface event f iff e includes f. (Note that e is a refinement of f.) Each module event can include at most one lower interface event.

M2. A is a refinement of B. Additionally, every event in A either satisfies the null image condition or it is the refinement of a single event in B. (Any invariant I_A that is used in proving the null image condition or the event refinement condition for events in A will have to be proved for A.) We say that module event e *corresponds* to upper interface event f iff e is a refinement of f.

M3. Every event in A corresponds to at most one interface event.

M4. Let module event e correspond to interface event f. If f is an input event of the upper interface or an output event of the lower interface, then f is *externally controlled*, that is, the execution of f causes the execution of e; otherwise, f is controlled by the module, and the execution of e causes the execution of f. For every externally-controlled interface event f, the following condition is required: there is an event e in A corresponding to f such that $possible(f) \Rightarrow enabled(e)$ is invariant for A.

M5. The specification of A satisfies the invariant requirements I_B, the progress requirements L_B, and any invariant requirements I_A needed for event refinement in **M2**, assuming that the fairness requirements of A, and the invariant requirements I_C and the progress requirements L_C of the lower interface are satisfied.

Condition **M1** ensures that for every behavior w of the state transition system A, its image in C is a behavior of the state transition system C. Condition **M2** ensures that for every behavior w of the state transition system A, its image in B is a behavior of the state transition system B. With **M1** and **M2**, every safety requirement that is implicitly specified by the state transition systems of the upper and lower interfaces is a safety property of the module. Condition **M3** is a straightforward way to ensure that no module event corresponds to more than one externally-controlled interface event. Condition **M4** ensures that the module is always ready to accept inputs from the upper interface and outputs from the lower interface. Consequently, if the progress requirements L_C are satisfied by interface C, they are guaranteed to hold for A and can be used in the proof specified by **M5**. (**M4** rules out a vacuous implementation of the module.)

Given **M1**—**M4**, to show that module A satisfies its interfaces B and C, our proof obligation is stated in **M5**. We assume the invariant and progress requirements of the lower interface (that is, the lower interface is offered by a correctly-implemented module at a lower level). We also assume that any actual implementation of module A satisfies the fairness requirements in its specification. Given these assumptions, **M5** requires a proof that module A satisfies (1) the invariant requirements I_B and progress requirements L_B of the upper interface, and (2) any invariant requirements I_A used in **M2**.

In formulating condition **M4**, we have made use of the invariant requirements of an interface. Thus **M4** is a weaker condition than requiring every module event that corresponds to an externally-controlled interface event to be always enabled, which is assumed in [13].

The above conditions can be generalized, in a straightforward manner, for specifying a module that does not use a lower interface or one that uses several lower interfaces. For a module that does not use a lower interface, we do not need **M1**; also, I_C and L_C are absent in **M5**. For a module that uses several lower interfaces, we require that the module satisfies **M1** for each lower interface, and the invariant and progress requirements of all lower interfaces are assumed in **M5**.

Lastly, we elaborate on how a module A can make use of a lower interface C that is offered by another module. Consider the constraint stated in **M1** that module A can access state variables of its lower interface C only via lower interface events. Given this constraint, the state variables of C are effectively not observable by A. Consequently, the state variables of C may be *auxiliary variables* in the module that offers interface C to A. For the same reason, the state variables of the upper interface B may be auxiliary variables in A.

3. Database Interface Specification

We define the following constants. Let OBJECTS denote the set of objects in a database, VALUES the set of values each object can have, KEYS a set of keys, and IDS a 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, given that each transaction has at most one procedure call outstanding. We will use *key*, *obj*, *val*, *id* as variables that range over the corresponding sets. For each *obj*, let its initial value be given by INITVALUE(*obj*).

We say that a transaction has a procedure invocation *outstanding* if it has called the procedure and the procedure has not yet returned. We say that the transaction is *active* if its *Begin* call has returned with a key, and the transaction has not yet ended.

3.1. State variables

H: sequence of $\{(id, Begin, key), (id, Read, key, obj, val), (id, Write, key, obj, val, \mathrm{OK}), (id, End, key, \mathrm{OK}), (id, Abort, key)\}$.
Initially, H is the null sequence.

History of the returns of procedure invocations. The $(id, Abort, key)$ entry is used to record every return that aborts a transaction. The other entries indicate successful returns. An unsuccessful *Begin* return is not recorded in H. H is adequate for stating serializability.

status(*id*): {NOTBEGUN, READY, COMMITTED, ABORTED} $\cup$ $\{(Begin), (Read, key, obj), (Write, key, obj, val), (End, key), (Abort, key)\}$.
Initially, *status*(*id*) = NOTBEGUN.

Indicating the status of transaction *id*. NOTBEGUN means that the transaction has not yet issued a *Begin* call, or such a call has returned with FAILED. READY means that the transaction is active and has no interface procedure invocation outstanding. A tuple, such as $(Read, key, obj)$, means that the transaction is active and has a procedure invocation outstanding as specified by the tuple. 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.

Notation: When we refer to a tuple in the domain of *status*(*id*), such as $(Read, key, obj)$, where a component in the tuple can have any of its allowed values, we shall omit that component in our reference. For example, *status*(*id*) = $(Read, obj)$ means *status*(*id*) = $(Read, key, obj)$ for some value of *key*. More than one component in a tuple may be omitted. For example, (obj) refers to $(Read, key, obj)$ for some *key* or $(Write, key, obj, val)$ for some *key* and some *val*. The same notational abbreviation is used in referring to elements of H. For example, $(id, obj) \in H$ means that H has a $(id, Read, obj, key, val)$ or a $(id, Write, obj, key, val, \mathrm{OK})$ entry for some *key* and some *val*.

3.2. State functions

active(*id*): boolean
True iff $(id, Begin) \in H$, and neither (id, End) nor $(id, Abort)$ is in H.

accessed(*id*): powerset of OBJECTS
The set of objects that have been accessed by transaction *id*.

$= \{obj\text{: } status(id) = (obj) \vee (id, obj) \in H\}$.

concurrentaccess (id): boolean

True iff there is an $i \in \text{IDS}-\{id\}$ such that transactions *id* and *i* have accessed a common object and were simultaneously active at some time in the past. Formally, it is true iff $accessed(i) \cap accessed(id)$ is not empty, and for some prefix h of H, $(id, Begin), (i, Begin) \in h$ and $(id, End), (i, End), (id, Abort), (i, Abort) \notin h$.

3.3. 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 its definition. We use @ to denote concatenation.

$Call(id, Begin) \equiv$
$\quad status(id) = \text{NOTBEGUN}$
$\quad \wedge\ status(id)' = (Begin)$

$Return(id, Begin, key) \equiv$
$\quad status(id) = (Begin) \wedge \neg allocated(key)$
$\quad \wedge\ status(id)' = \text{READY}$
$\quad \wedge\ allocated(key)'$
$\quad \wedge\ H' = H @ (id, Begin, key)$

$Return(id, Begin, \text{FAILED}) \equiv$
$\quad status(id) = (Begin)$
$\quad \wedge\ status(id)' = \text{NOTBEGUN}$

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

$Return(id, Read, key, obj, val) \equiv$
$\quad status(id) = (Read, key, obj)$
$\quad \wedge\ status(id)' = \text{READY}$
$\quad \wedge\ H' = H @ (id, Read, key, obj, val)$

$Return(id, Read, key, obj, \text{ABORT}) \equiv$
$\quad status(id) = (Read, key, obj) \wedge concurrentaccess(id)$
$\quad \wedge\ status(id)' = \text{ABORTED}$
$\quad \wedge\ \neg allocated(key)'$
$\quad \wedge\ H' = H @ (id, Abort, key)$

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

$Return(id, Write, key, obj, val, \text{OK}) \equiv$
$\quad status(id) = (Write, key, obj, val)$
$\quad \wedge\ status(id)' = \text{READY}$
$\quad \wedge\ H' = H @ (id, Write, key, obj, val, \text{OK})$

$Return(id, Write, key, obj, val, \text{ABORT}) \equiv$
$\quad status(id) = (Write, key, obj, val) \wedge concurrentaccess(id)$
$\quad \wedge\ status(id)' = \text{ABORTED}$
$\quad \wedge\ \neg allocated(key)'$
$\quad \wedge\ H' = H @ (id, Abort, key)$

$Call(id, End, key) \equiv$
$\quad status(id) = \text{READY} \wedge allocated(key)$
$\quad \wedge\ status(id)' = (End, key)$

$Return(id, End, key, \text{OK}) \equiv$
$\quad status(id) = (End, key)$
$\quad \wedge\ status(id)' = \text{COMMITTED}$
$\quad \wedge\ \neg allocated(key)'$
$\quad \wedge\ H' = H @ (id, End, key, \text{OK})$

$Return(id, End, key, \text{ABORT}) \equiv$
$\quad status(id) = (End, key) \wedge concurrentaccess(id)$
$\quad \wedge\ status(id)' = \text{ABORTED}$
$\quad \wedge\ \neg allocated(key)'$
$\quad \wedge\ H' = H @ (id, Abort, key)$

$Call(id, Abort, key) \equiv$
$\quad status(id) = \text{READY} \wedge allocated(key)$
$\quad \wedge\ status(id)' = (Abort, key)$

$Return(id, Abort, key) \equiv$
$\quad status(id) = (Abort, key)$
$\quad \wedge\ status(id)' = \text{ABORTED}$
$\quad \wedge\ \neg allocated(key)'$
$\quad \wedge\ H' = H @ (id, Abort, key)$

3.4. Safety requirements

One safety requirement is that each transaction can only issue a correct sequence of procedure calls. Specifically, the following state function is invariant for the interface:

legal(*id*): boolean

True iff the subsequence of (*id*) entries in H is a prefix of $(id, Begin)$@<successes>@<final>, where <successes> is a sequence of zero or more (id, obj) entries, and <final> is either $(id, Abort)$ or (id, End).

It can be shown that *legal*(*id*) is invariant for the interface state transition system above (proof omitted). Thus, this first safety requirement has been specified by the state transition system and does not have to be explicitly stated.

Let us point out some other safety requirements that are implicit in the state transition system. An invocation of *Begin* is always enabled to return FAILED. (In an actual implementation, an invocation of *Begin* returns FAILED only when there are insufficient resources to start another transaction, e.g., when there is no unallocated key.)

An invocation of *Read*, *Write*, or *End* by transaction *id* aborts only if it has accessed an object that has been accessed by another transaction, one that was concurrently active at some time in the past.

This requirement has been specified by including *concurrentaccess*(id) in the enabling conditions of the corresponding return events. (For a single-version implementation, this condition can be strengthened by requiring both id and i to be currently active.)

The only invariant requirement of the interface to be explicitly specified is serializability. Before defining serializability, we need to define some more notation. For any sequence h, we use h_i to denote the ith element of h, $h_{<i}$ to denote the prefix of h up to but excluding h_i, and $h_{\leq i}$ to denote the prefix of h up to and including h_i. For any id, $H(id)$ denotes the subsequence of H obtained from it by including only the (id) entries.

For any obj and any sequence h of transaction returns, define

lastvalue (obj, h): VALUES
: =INITVALUE(obj), if $(obj) \notin h$.
: $=val$, if $(obj) \in h$ and (obj, val) is the last such entry.

Definition: H is *serializable* iff there is a permutation $id_1, id_2, \cdots, id_n$ of all the committed transactions such that $S = H(id_1)@H(id_2)@ \cdots @H(id_n)$ satisfies

$$S_i=(Read, obj, val) \Rightarrow val=lastvalue(obj, S_{<i}).$$

Define the following state functions:

keyof(id): KEYS $\cup$ {NULL}
: =NULL, if $\neg active(id)$.
: $=key$, if $active(id)$ and $(id, Begin, key)$ in H.

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

The interface specification includes the following:

Invariant requirement: *correctkeyuse* $\Rightarrow H$ is serializable.

A comment on the above definition of serializability is in order. We find three definitions of serializability in [3]: *conflict serializability*, *view serializability*, and *multi-version view serializability*. The first two are applicable to single-version implementations. The two-phase locking implementation satisfies conflict serializability, the strongest condition of the three. However, the multi-version timestamp implementation satisfies only multi-version view serializability, the weakest condition of the three. The above definition is a statement of multi-version view serializability, which is the only condition of the three that can be used for both implementations to be specified in this paper.

3.5. Progress requirements

A progress requirement specifying that every procedure call eventually returns is this:

$$L_1 \equiv \quad status(id) \in \{(Begin), (Read), (Write), (End), (Abort)\}$$
$$leads\text{-}to\ status(id) \in \{\text{READY, ABORTED, COMMITTED, NOTBEGUN}\}$$

Lamport's assumption that if a transaction is not aborted, then the transaction is eventually terminated (by its client program) with an invocation of *End*, can be stated as follows: If every *Read* and *Write* call made by the transaction returns successfully, then the transaction eventually issues an *End* call. Formally:

$L_2 \equiv$ $(status(id) \in \{(Read), (Write)\}$ *leads-to* $status(id)$=READY)
$\Rightarrow (status(id)$=READY *leads-to* $status(id)=(End))$

The interface specification includes the following:

Progress requirement: L_1 holds, assuming that *correctkeyuse* is invariant and L_2 holds.

4. Database Implementation Using Two-Phase Locking

The two-phase locking implementation is built on top of a physical database. The interface offered by the physical database is the lower interface used by the two-phase locking implementation. In Section 4.1, we specify the lower interface. In Section 4.2, we specify the module that implements the two-phase locking protocol such that it satisfies module implementation conditions **M1**—**M4**. In Sections 4.3 and 4.4, we show that it satisfies the proof obligation in **M5**.

4.1. Lower interface specification

Note that outstanding procedure calls at the lower interface for accessing the physical database are uniquely identified by the entries of KEYS.

State variables

$status_L(key)$: {READY, $(AcqLock, obj)$, $(RelLock, obj)$, $(Read_L, obj)$, $(Write_L, obj, val)$}.
Initially READY.

Indicating the status of any procedure invocation identified by *key*. READY means that *key* has no procedure invocation outstanding at the lower interface. Otherwise, the outstanding procedure invocation is indicated by a tuple.

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

True iff *key* has locked *obj*.

storedvalue (obj): VALUES. Initially, INITVALUE(obj).

The value of *obj* in the physical database.

State functions

waiting (key, obj): boolean.

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

waitfor graph:

Directed graph defined by nodes KEYS $\cup$ OBJECTS and
edges $\{(x,k)\text{: } owned(k,x)\} \cup \{(k,x)\text{: } waiting(k,x)\}$.

$cycle(k_1, k_2, \cdots, k_i)$: boolean.

True iff keys $k_1, k_2, \cdots, k_i$ form a cycle in *waitfor graph*, that is, there exist objects $x_1, x_2, \cdots, x_i$ such that $waiting(k_j, x_j) \wedge owned(k_{j+1}, x_j)$ for $1 \le j < i$, and $waiting(k_i, x_1) \wedge owned(k_1, x_1)$.

deadlock (key, obj): boolean.

True iff there is a cycle including the edge (key, obj) in *waitfor graph*.

Events

The interface events are the calls and returns of the interface procedures *AcqLock*, *RelLock*, $Read_L$, and $Write_L$.

$$
\begin{aligned}
&Call(key, AcqLock, obj) \equiv \\
&\qquad status_L(key){=}\mathrm{READY} \\
&\qquad \wedge\; status_L(key)'{=}(AcqLock, obj)
\end{aligned}
$$

$$
\begin{aligned}
&Return(key, AcqLock, obj, \mathrm{GRANTED}) \equiv \\
&\qquad status_L(key){=}(AcqLock, obj) \wedge [\forall k: \neg owned(k, obj)] \\
&\qquad \wedge\; status_L(key)'{=}\mathrm{READY} \\
&\qquad \wedge\; owned(key, obj)'
\end{aligned}
$$

$$
\begin{aligned}
&Return(key, AcqLock, obj, \mathrm{REJECTED}) \equiv \\
&\qquad status_L(key){=}(AcqLock, obj) \wedge deadlock(key, obj) \\
&\qquad \wedge\; status_L(key)'{=}\mathrm{READY}
\end{aligned}
$$

$$
\begin{aligned}
&Call(key, RelLock, obj) \equiv \\
&\qquad status_L(key){=}\mathrm{READY} \\
&\qquad \wedge\; status_L(key)'{=}(RelLock, obj)
\end{aligned}
$$

$$
\begin{aligned}
&Return(key, RelLock, obj) \equiv \\
&\qquad status_L(key){=}(RelLock, obj) \wedge owned(key, obj) \\
&\qquad \wedge\; status_L(key)'{=}\mathrm{READY} \\
&\qquad \wedge\; \neg owned(key, obj)'
\end{aligned}
$$

$$
\begin{aligned}
&Call(key, Read_L, obj) \equiv \\
&\qquad status_L(key){=}\mathrm{READY} \\
&\qquad \wedge\; status_L(key)'{=}(Read_L, obj)
\end{aligned}
$$

$$
\begin{aligned}
&Return(key, Read_L, obj, val) \equiv \\
&\qquad status_L(key){=}(Read_L, obj) \\
&\qquad \wedge\; status_L(key)'{=}\mathrm{READY} \\
&\qquad \wedge\; val{=}storedvalue(obj)
\end{aligned}
$$

$$
\begin{aligned}
&Call(key, Write_L, obj, val) \equiv \\
&\qquad status_L(key){=}\mathrm{READY} \\
&\qquad \wedge\; status_L(key)'{=}(Write_L, obj, val)
\end{aligned}
$$

$$
\begin{aligned}
&Return(key, Write_L, obj, val) \equiv \\
&\qquad status_L(key){=}(Write_L, obj, val) \\
&\qquad \wedge\; status_L(key)'{=}\mathrm{READY} \\
&\qquad \wedge\; storedvalue(obj)'{=}val
\end{aligned}
$$

Safety requirements

Safety requirements of the lower interface are all implicitly specified by the state transition system. 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.

Progress requirements

The physical database that offers the lower interface guarantees progress properties Q_1 through Q_5:

$Q_1 \equiv$ $status_L(key)$=$(Read_L)$ *leads−to* $status_L(key)$=READY

$Q_2 \equiv$ $status_L(key)$=$(Write_L)$ *leads−to* $status_L(key)$=READY

$Q_3 \equiv$ $status_L(key)$=$(RelLock, obj) \wedge owned(key, obj)$
leads−to $status_L(key)$=READY $\wedge \neg owned(key, obj)$

$Q_4 \equiv$ G_4 holds, assuming that R_4 holds
where

$R_4 \equiv [\forall k_2\colon waiting(k_1, obj) \wedge owned(k_2, obj)$ *leads−to* $waiting(k_1, obj) \wedge \neg owned(k_2, obj)]$

$G_4 \equiv waiting(k_1, obj)$ *leads−to* $owned(k_1, obj)$

Q_4 specifies the property that every call to *AcqLock* eventually returns successfully provided that every granted lock is eventually returned and the caller continues to wait for the lock (i.e., is not aborted). In other words, if *Return*(*key*, *AcqLock*, *obj*, GRANTED) is enabled infinitely often, it eventually occurs. This is how we interpret Lamport's statement that the interface does not starve an individual process [12].

$Q_5 \equiv$ $cycle(k_1, k_2, \cdots, k_n)$ *leads−to* $[\exists k_i, 1 \le i \le n\colon status_L(k_i)$=READY$]$

Q_5 specifies that if there is a cycle of deadlocked processes, it is eventually broken.

4.2. Two-phase locking implementation

The two-phase locking implementation is obtained from the database interface by adding state variables, refining the database interface events, and adding new events. The new events are obtained by refining events of the lower interface.

State variables

In addition to the state variables *H*, *status*, and *allocated* in the database interface specification, and the state variables $status_L$, *owned*, and *storedvalue* in the lower interface, we add the following:

locked(*key*, *obj*): boolean. Initially false.
True iff *key* has locked *obj*.

localvalue(*obj*, *key*): VALUES $\cup$ {NULL}. Initially NULL.
Current value of *obj* as seen by transaction using *key*.

aborting(*key*): boolean. Initially false.
True iff the transaction using *key* has been rejected in acquiring a lock and it has not yet aborted.

S: sequence of {(*id*, *Begin*, *key*), (*id*, *Read*, *key*, *obj*, *val*), (*id*, *Write*, *key*, *obj*, *val*, OK), (*id*, *End*, *key*, OK)}.
Initially, *S* is the null sequence.

An auxiliary variable. A serial history obtained by concatenating the histories of committed transactions in the order of commitment.

The state variable H in the database interface specification becomes an auxiliary variable. This also makes auxiliary all state functions defined in terms of H, such as *concurrentaccess*, etc. Recall that the values of auxiliary variables and functions cannot affect the enabling conditions of events nor can they affect the update of a nonauxiliary variable.

State functions

holdinglocks (*key*): boolean.
True iff *locked* (*key*, *obj*) is true for some *obj*.

Events

Implementation events that are obtained by refining database interface events are listed first. In an event formula, we use <previous definition> to denote the formula defining the corresponding event 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.

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

$Return(id, Begin, key) \equiv$ <previous definition> $\wedge \neg holdinglocks(key)$

$Return(id, Begin, \text{FAILED}) \equiv$ <previous definition>

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

$Return(id, Read, key, obj, val) \equiv$ <previous definition> $\wedge\ localvalue(obj, key) \neq \text{NULL}$
$\wedge\ val = localvalue(obj, key)$

$Return(id, Read, key, obj, \text{ABORT}) \equiv$
$status(id) = (Read, key, obj) \wedge aborting(key)$
$\wedge\ status(id)' = \text{ABORTED}$
$\wedge\ H' = H @ (id, Abort, key)$
$\wedge\ \neg allocated(key)'$
$\wedge\ \neg aborting(key)'$
$\wedge\ [\forall x:\ localvalue(x, key)' = \text{NULL}]$

For the above to be a refinement of the corresponding interface event, it is sufficient that *concurrentaccess* (*id*) is true whenever *status* (*id*)=(*obj*) and *aborting* (*key*) are true. The following safety requirement is adequate:

$A_1 \equiv \quad keyof(id) = key \wedge status(id) = (obj) \wedge aborting(key) \Rightarrow concurrentaccess(id)$

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

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

$Return(id, Write, key, obj, val, \text{ABORT}) \equiv$
$\quad status(id)=(Write, key, obj, val) \wedge aborting(key)$
$\quad \wedge\ status(id)'=\text{ABORTED}$
$\quad \wedge\ H'=H@(id, Abort, key)$
$\quad \wedge\ \neg allocated(key)'$
$\quad \wedge\ \neg aborting(key)'$
$\quad \wedge\ [\forall x:\ localvalue(x, key)'=\text{NULL}]$

A_1 ensures that the above event satisfies the event refinement condition.

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

$Return(id, End, key, \text{OK}) \equiv$ <previous definition> $\wedge\ [\forall x:\ localvalue(x, key)=\text{NULL}]$
$\quad \wedge\ S'=S@H(id)$

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

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

$Return(id, Abort, key) \equiv$ <previous definition>
$\quad \wedge\ [\forall x:\ localvalue(x, key)'=\text{NULL}]$

In addition to the above events obtained by refining database interface events, the following events are obtained by refining lower interface events. These events have null images at the database interface because they do not update any state variables of the database interface.

$RequestLock(id, key, obj) \equiv$
$\quad status(id) \in \{(Read, key, obj), (Write, key, obj)\} \wedge \neg locked(key, obj)$
$\quad \wedge\ Call(key, AcqLock, obj)$

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

$LockRejected(key, obj) \equiv$
$\quad Return(key, AcqLock, obj, \text{REJECTED})$
$\quad \wedge\ aborting(key)'$

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

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

$RequestWrite(id, key, obj) \equiv$
$\quad status(id)=(End, key) \wedge localvalue(obj, key) \neq \text{NULL}$
$\quad \wedge\ Call(key, Write_L, obj, localvalue(obj, key))$

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

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

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

It can be easily checked that module implementation conditions **M1**—**M4** are satisfied. In particular, **M4** is satisfied for every call event of the database interface because the corresponding implementation event is exactly the same. **M4** is satisfied for every return event of the lower interface because each such event f appears in an implementation event of the form $f \wedge p$ and $enabled(p)$ is true. Condition **M5** will be established in Sections 4.3 and 4.4.

We now give an informal description of the two-phase locking implementation, by indicating the sequence of event occurrences for each transaction call. (Those who are familiar with the two-phase locking protocol might want to go ahead to Section 4.3.) For brevity, we will omit parameters in event names whenever the omission results in no ambiguity.

Suppose a client program begins a new transaction by issuing a $Call(Begin)$. Eventually the implementation executes either $Return$(FAILED) or $Return(key)$. In the former case, the transaction's execution is over. In the latter case, read or write calls can be issued for the transaction, and the transaction enters its *growing* stage.

Suppose a $Call(Write, key, obj, val)$ is issued, where *obj* has been previously accessed by the transaction. Then *obj* is locked by *key*. The implementation assigns *val* to $localvalue(obj, key)$ and executes $Return(Write, \text{OK})$

Suppose a $Call(Write, key, obj, val)$ is issued, where *obj* has not yet been accessed by the transaction. Then *obj* is not locked by *key*. The implementation executes $RequestLock(key, obj)$. Eventually the lower interface returns causing either $LockAcquired(key, obj)$ or $LockRejected(key, obj)$ to occur. In the first case, the implementation sets $localvalue(obj, id)$ to *val* and executes $Return(Write, \text{OK})$. The second case will be considered below.

Suppose a $Call(Read, key, obj)$ is issued, where *obj* has been previously accessed by the transaction. The implementation executes $Return(Read, val)$ where *val* equals $localvalue(obj, key)$.

Suppose a $Call(Read, key, obj)$ is issued, where *obj* has not been previously accessed by the transaction. As in the case of the write above, the implementation executes $RequestLock(key, obj)$, which is eventually followed by either $LockAcquired(key, obj)$ or $LockRejected(key, obj)$. In the first case, the implementation executes $RequestRead(key, obj)$. Eventually a return from the lower interface causes $ReadCompleted(obj, val)$ to occur. At this point, *val*, which equals $storedvalue(obj)$, is assigned to $localvalue(obj, id)$. After this, the implementation executes $Return(Read, obj, val)$.

Suppose a $Call(End, key)$ is issued by the client program. The transaction goes through two stages of activity. In the first stage, referred to as *committing*, the local value of each object accessed by the transaction is written into the physical store. Specifically, for each *obj* with $localvalue(obj, key) \neq \text{NULL}$, there is an occurrence of $RequestWrite(key, obj)$ which is followed by an occurrence of $WriteCompleted(key, obj)$. When all the local values have been written to the physical database, the implementation executes a *Return*(*End*, OK), ending the transaction's execution. The second stage, referred to as *lock-releasing*, then follows. In this stage, the implementation returns all the locks acquired by the transaction. For each *obj* such that *locked*(*key*, *obj*) is true, the implementation executes a *ReqRelLock*(*key*, *obj*), which is followed by the occurrence of *LockReleased*(*key*, *obj*). The second stage ends when all the locks are returned. The key can now be reallocated to a new transaction.

Two cases have not yet been considered: a *Call*(*Abort*) issued for the transaction, and the occurrence of *LockRejected*(*key*, *obj*) following a *RequestLock*(*key*, *obj*). In each case, the implementation returns the locks acquired by the transaction, exactly as in the lock-releasing stage following a *Call*(*End*, *key*).

4.3. Proof of safety

We prove that the above implementation satisfies the following invariant requirements, assuming that *correctkeyuse* is invariant:

$A_1 \equiv \quad keyof(id)=key \wedge status(id)=(obj) \wedge aborting(key) \Rightarrow concurrentaccess(id)$

$A_2 \equiv \quad S_i=(Read, obj, val) \Rightarrow val=lastvalue(obj, S_{<i})$.

A_1 ensures that the implementation events are refinements of the database interface events. A_2 ensures serializability. Recall that S is a serial history obtained by concatenating histories of committed transactions in the order of commitment. We provide here an informal justification of the invariance of A_1 and A_2. A formal proof is given in Appendix A.

Consider A_1. Assume $keyof(id)=key \wedge status(id)=(obj)$. When transaction *id* becomes active, *aborting*(*key*) is false. It is set to true only in the *LockRejected* event, when the lower interface executes *Return*(*key*, *AcqLock*, *obj*, REJECTED). The latter occurs only if *deadlock*(*key*, *obj*) is true. From the definition of *deadlock*, we have a cycle in the *waitfor graph* involving the edge (*key*, *obj*). Thus, *owned*(*k*, *obj*) is true for some key $k \neq key$ that is allocated to a transaction i. Since transaction i is also waiting for a key, it is active. Additionally, *obj* belongs to *accessed*(i). From $status(id)=(obj)$, we know that transaction *id* is active and *obj* belongs to *accessed*(*id*). Thus, *concurrentaccess*(*id*) holds just before *aborting*(*key*) becomes true. Once *concurrentaccess*(*id*) holds, it is obvious from its definition that it never becomes false.

Consider A_2. The *Return*(*id*, *End*, OK) event is the only event that can affect A_2. It concatenates $H(id)$ to the end of S. Thus A_2 is invariant if the following is invariant:

$A_3 \equiv \quad active(id) \wedge H(id)_i=(Read, obj, val) \Rightarrow$
(a) $\quad ((obj) \notin H(id)_{<i} \Rightarrow val=lastvalue(obj, S)) \wedge$
(b) $\quad ((obj) \in H(id)_{<i} \Rightarrow val=lastvalue(obj, H(id)_{<i}))$

To establish that A_3 is invariant, we need to relate several values associated with each object: i.e., its stored value, its last value in S, and, whenever it is locked by a transaction, its local value and last value in H. The following invariant assertions relate these values during the growing and committing stages of a transaction:

$A_4 \equiv \quad (\forall key: \neg locked(key, obj)) \Rightarrow storedvalue(obj)=lastvalue(obj, S)$

$A_5 \equiv \quad keyof(id)=key \wedge \neg locked(key, obj) \Rightarrow (id, obj) \notin H \wedge localvalue(obj, key)=\text{NULL}$

$A_6 \equiv \quad keyof(id)=key \wedge locked(key, obj) \wedge status(id) \neq (End) \Rightarrow$
$\quad storedvalue(obj)=lastvalue(obj, S) \wedge$

(a) $[((id, obj) \notin H \wedge localvalue(obj, key){=}\mathrm{NULL}) \vee$

(b) $((id, obj) \notin H \wedge localvalue(obj, key){=}lastvalue(obj, S)) \vee$

(c) $((id, obj) \in H \wedge localvalue(obj, key){=}lastvalue(obj, H(id)))]$

$A_7 \equiv$ $keyof(id){=}key \wedge locked(key, obj) \wedge status(id){=}(End) \Rightarrow$

$(id, obj) \in H \wedge$

(a) $[(localvalue(obj, key){=}lastvalue(obj, H(id)) \wedge storedvalue(obj){=}lastvalue(obj, S))$

(b) $\vee (localvalue(obj, key){=}\mathrm{NULL} \wedge storedvalue(obj){=}lastvalue(obj, H(id)))]$

A_4 states that when an object is not locked by any transaction, its stored value is its last value in S. This is true initially, when both are equal to the initial value of the object. This is preserved whenever the stored value is changed, because a change happens only when a transaction has locked the object *and* is in the committing stage. When the transaction commits, S is updated and the consequent of A_4 is established. And A_4 is preserved when the lock is released subsequently.

A_5 is invariant because a transaction reads or writes an object only after it has locked the object.

A_6 is about an object locked by a transaction that is in its growing stage. The first conjunct in the consequent states that its stored value equals its last value in S. This holds when the transaction first acquires the lock on the object (by A_4). It holds subsequently because this transaction is not in its committing stage, and because no other transaction can change its stored value while this transaction has a lock on it. The second conjunct in the consequent relates the local value of the object to its last value in H. Disjunct (a) holds just after the transaction has locked the object, when its local value is NULL. If the transaction's first access to the object is a *Read*, disjunct (b) holds after the stored value has been retrieved into the local value but before *Return*(*Read*) occurs. Disjunct (c) holds after the successful return of a read or write call.

A_7 is about an object locked by a transaction that is in the committing stage. In the consequent, the first conjunct states that the object has been accessed by the transaction. The second conjunct relates its local value, its last value in $H(id)$, its stored value, and its last value in S. Disjunct (a) holds just after the transaction has invoked *Call*(*End*) because of A_6; note that at this point S does not yet include $H(id)$. Disjunct (b) holds after the local value has been written into the stored value. Also, when *Return*(*End*, OK) occurs, A_4 is established for this object because of disjunct (b).

We use the notation A_{i-j} to denote $A_i \wedge A_{i+1} \wedge \cdots \wedge A_j$.

Lemma 1. $A_1 \wedge A_{4-7}$ is invariant, assuming that *correctkeyuse* is invariant. (Proof in Appendix A.)

It remains for us to prove that A_2 is invariant, which ensures serializability of the database interface. Recall that it is sufficient to prove A_3 to be invariant. By Lemma 1, we can make use of the result that A_5 and A_6 are invariant in our proof. From A_5, observe that an object is accessed by transaction *id* only if the transaction has locked it. Thus, the consequent of A_6 holds just prior to the occurrence of *Return*(*id*, *Read*, *obj*, *val*). There are two cases. If $(obj) \notin H(id)$ holds prior to the occurrence, then we have $val = lastvalue(obj, S)$, by A_6(b). If $(obj) \in H(id)$ holds prior to the occurrence, then we have $val = lastvalue(obj, H(id))$, by A_6(c). In each case, the consequent of A_3 holds after the occurrence.

4.4. Proof of progress

The following fairness requirement for events of the two-phase locking module implementation is assumed:

F1: Every event that is not a call event has weak fairness.

Progress requirements Q_1, Q_2, Q_3, Q_4, Q_5 are assumed to hold for the lower interface. By **M4**, Q_1 through Q_5 also hold for the module implementation We proceed to prove that the module implementation satisfies the progress requirement of the database interface.

Lemma 2. The following progress assertions hold for the module implementation:

$W_1 \equiv$ $status(id)=(Begin)$ *leads-to* $status(id) \in \{\text{READY, NOTBEGUN}\}$

$W_2 \equiv$ $status(id)=(End, key)$ *leads-to* $status(id)=\text{COMMITTED}$

$W_3 \equiv$ $status(id)=(Abort)$ *leads-to* $status(id)=\text{ABORTED}$

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

$W_5 \equiv$ $status(id)=(key, obj) \wedge \neg locked(key, obj)$ *leads-to* $waiting(key, obj)$

$W_6 \equiv$ $status(id)=(key, obj) \wedge aborting(key)$ *leads-to* $status(id)=\text{ABORTED}$

Proof:

W_1 holds as follows. The state formula $status(id)=(Begin)$ can only be falsified by $Return(id, Begin, \text{FAILED})$ and by $Return(id, Begin, key)$ for some *key*. The occurrence of the latter establishes $status(id)=\text{READY}$. The former is continuously enabled, and its occurrence establishes $status(id)=\text{NOTBEGUN}$.

W_2 holds as follows. From Q_2, *RequestWrite*, and *WriteCompleted*, we have:

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

No event can falsify $localvalue(obj, key)=\text{NULL}$ while $status(id)=(End, key)$. Therefore, from the above we have:

$$status(id)=(End, key)\ leads\text{-}to$$
$$status(id)=(End, key) \wedge (\forall obj: localvalue(obj, key)=\text{NULL})$$

From $Return(End, key)$, we have:

$$status(id)=(End, key) \wedge (\forall obj: localvalue(obj, key)=\text{NULL})$$
$$leads\text{-}to\ status(id)=\text{COMMITTED}$$

Combining the above two, we have W_2.

W_3 holds from $Return(Abort)$.

W_4 holds as follows. From Q_1, *RequestRead* and *ReadCompleted*, we have:

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

From above and $Return(Read, val)$, we have:

$$status(id)=(Read, key, obj) \wedge locked(key, obj)\ leads\text{-}to\ status(id)=\text{READY}$$

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

$$status(id)=(Write, key, obj) \wedge locked(key, obj)\ leads\text{-}to\ status(id)=\text{READY}$$

Combining the above two, we have W_4.

W_5 holds from *RequestLock*.

W_6 holds from *Return* (*Read*, *key*, *obj*, ABORT) and *Return* (*Write*, *key*, *obj*, ABORT).
End of proof.

The events that falsify *waiting* (*key*, *obj*) establish either *locked* (*key*, *obj*) or *aborting* (*key*). Therefore, from W_1, W_2, W_3, W_4, W_5, W_6, all that is left to establish the desired progress property is:

$W_7 \equiv$ *waiting* (k_1, obj) *leads−to* ¬*waiting* (k_1, obj)

where we have used k_1 in place of *key* for notational convenience.

We prove W_7 using a lexicographically ordered metric on the *waitfor graph*. Recall that the *waitfor graph* is the directed graph defined on by nodes KEYS ∪ OBJECTS and edges $\{(x,k)\colon owned(k,x)\} \cup \{(k,x)\colon waiting(k,x)\}$. Note that there is no edge from a key to a key, or from an object to an object. Every node can have at most one outgoing edge. Because k_1 is waiting, it has an outgoing edge.

Consider the succession of nodes on the path starting from k_1. Let $k_1, x_1, k_2, x_2, \cdots, k_J$ be the sequence of *distinct* nodes such that *waiting* (k_i, x_i) and *owned* (k_{i+1}, x_i) for $1 \le i < J$, and k_J satisfies one of the following three mutually exclusive conditions:

unblocked(k_J) ≡ k_J is not waiting for any object.

blocked(k_J) ≡ k_J is waiting for an object that is not locked by any key.

deadlocked(k_J) ≡ k_J is waiting for an object locked by k_i for some i, $1 \le i < J$.

While k_1 is waiting, this path can grow and shrink. We need to prove that eventually this path shrinks to only k_1. Observe that J is a state function indicating the number of keys in the path, and k_J indicates the last key on the path. At any time, $1 \le J \le |\text{KEYS}|$. Every k_i, $1 \le i < J$, is waiting and hence allocated to some transaction. The last key k_J can be either allocated or not allocated. It is not allocated if k_J is in the stage of releasing locks acquired when it was last allocated and has not yet released the lock on x_{J-1}.

Define the following functions, where $1 \le i \le |\text{KEYS}|$. We assume that $|\text{OBJECTS}| < M$, for some arbitrary but fixed integer M.

α_i: integer
 = number of objects locked by k_i, if $i < J$, or $i = J$ and *allocated* (k_J).
 $= M$, if $i = J$ and ¬*allocated* (k_J).
 $= 0$, if $i > J$.

β_i: integer
 = number of times x_i has been unlocked since k_i last started to wait,
  if $i < J$, or $i = J$ and k_J is waiting.
 $= 0$, if $i > J$, or if $i = J$ and k_J is not waiting.

Define the function $\Delta = (\beta_1, \alpha_2, \beta_2, \alpha_3, \cdots, \alpha_{|\text{KEYS}|}, \beta_{|\text{KEYS}|})$. The values of Δ are well-ordered lexicographically. We shall prove the following:

$W_8 \equiv$ *waiting* $(k_1, x_1) \wedge \Delta = a$ *leads−to* ¬*waiting* $(k_1, x_1) \vee \Delta > a$

We first show that W_7 follows from W_8. W_8 states that that Δ increases without bound unless k_1 stops waiting. For Δ to increase without bound, either β_i or α_i must increase without bound for some i. The former is not allowed by Q_4, which says that the lock manager in the physical database is fair. The latter is not allowed by L_2, the assumption that every transaction accesses at most a finite number of objects. Thus, it suffices to prove W_8.

Lemma 3. The following progress assertion holds:

$W_9 \equiv$ *unblocked*$(k_J) \wedge \Delta{=}a$ *leads-to* $W_{9a} \vee W_{9b} \vee W_{9c}$, where

$W_{9a} \equiv$ *unblocked*$(k_J) \wedge \Delta{>}a$

$W_{9b} \equiv$ *blocked*$(k_J) \wedge \Delta{\geq}a$

$W_{9c} \equiv$ *deadlocked*$(k_J) \wedge \Delta{\geq}a$

Proof.
Assume $J{=}j$ and $\neg$*allocated*(k_J), that is, k_j is releasing its locks. $J{=}j$ and $\Delta{=}a$ hold until k_j releases its lock on x_{j-1}. At this point, W_{9b} holds with $J{=}j-1$ and $\Delta{>}a$. α_j decreases from M to 0, and β_{j-1} increases by 1. No other α_i or β_i is affected. Δ increases because β_{j-1} is lexicographically more significant than α_j.

Assume $J{=}j$ and *allocated*(k_J). Eventually the transaction using k_j requests an abort, a commit, or an access to an object not previously accessed by it (by L_2 and W_4). If the transaction requests an abort or a commit, k_j eventually becomes deallocated (by W_2 and W_3). When this happens, α_j becomes M and J remains j. Thus, Δ increases and we have W_{9a}.

Suppose $J{=}j$ and k_j requests access to an object not previously accessed by it. If the object is not locked, then W_{9b} holds with $J{=}j$ and $\Delta{=}a$. If the object is locked by some key already on the path, (that is, by k_i for some i, $1{\leq}i{<}j$), then W_{9c} results with $J{=}j$ and $\Delta{=}a$. If the object is locked by some key not already on the path, then the path gets extended, resulting in $\Delta{>}a$; specifically, J becomes $l{>}j$, and α_i increases from 0 for $j{<}i{\leq}l$. W_{9a} holds if *unblocked*(k_l). W_{9b} holds if *blocked*(k_l). W_{9c} holds iff prior to the request by k_j, k_l was a predecessor to a key on the path.
End of proof.

Lemma 4. The following progress assertion holds:

$W_{10} \equiv$ *blocked*$(k_J) \wedge \Delta{=}a$ *leads-to* *unblocked*$(k_1) \vee \Delta{>}a$

Proof. Assume $J{=}j$, and let k_j be waiting for x_j. The *LockAcquired*(k_j, x_j) event is continuously enabled while *blocked*(k_j), and it eventually occurs unless some other key locks x_j. Assume the former case: that is, k_j locks x_j. If $j{=}1$, we get *unblocked*(k_1). If $j{>}1$, we get $\Delta{>}a$ because α_j increases by 1. In either case, the value of J remains to be j. Next assume that x_j is locked by a key other than k_j. We get $\Delta{>}a$ because J becomes $j+1$ and α_{j+1} increases from 0.
End of proof.

Lemma 5. The following progress assertion holds:

$W_{11} \equiv$ *deadlocked*$(k_J) \wedge \Delta{=}a$ *leads-to* *unblocked*$(k_1) \vee \Delta{>}a$

Proof. Assume $J{=}j$. Then we have a cycle consisting of k_j and other (perhaps all) keys in the path. *LockRejected*(k_i, x_i) is enabled for every k_i in the cycle, and eventually the lock manager in the physical database executes one of them (by Q_5). Suppose *LockRejected*(k_l, x_l) occurs, for some $1{\leq}l{\leq}j$. If $l{=}1$, we get *unblocked*(k_1). If $l{>}1$, then J becomes l, α_i and β_i become 0 for $l{<}i{\leq}j$, and α_l increases to M. $\Delta{>}a$ holds because α_l is lexicographically more significant than α_i or β_i, for $l{<}i{\leq}j$.
End of proof.

Applying the transitivity and disjunction rules to W_9, W_{10}, and W_{11} (with W_{10} at W_{9b}, and W_{11} at W_{9c}), we get $\mathit{unblocked}(k_J) \wedge \Delta{=}a$ *leads−to* $\mathit{unblocked}(k_1) \vee \Delta{>}a$. Applying the disjunction rule to this, W_{10}, and W_{11}, we get W_8, noting that $\mathit{unblocked}(k_1) \Rightarrow \neg \mathit{waiting}(k_1, x_1)$ and $\mathit{unblocked}(k_J) \vee \mathit{blocked}(k_J) \vee \mathit{deadlocked}(k_J) \equiv \mathit{true}$. Recall that W_8 is sufficient for W_7, and W_1—W_7 are sufficient for the module implementation to satisfy the progress requirement of the database interface.

5. Multi-Version Timestamp Implementation

The multi-version timestamp module implementation is obtained from the database interface by adding state variables, and refining the database interface events. Unlike the two-phase locking implementation, there is no lower interface.

The implementation uses ''timestamps" that are nonnegative integers. To be consistent with the database interface specification, timestamps will be referred to as keys. For each object, the implementation maintains multiple versions, one for each transaction that has written into the object and has not yet aborted. Each version ov is a record with three components: $ov.wkey$, the key of the transaction that wrote the version; $ov.value$, the value that was written; and $ov.rkey$, the largest key among keys of transactions that have read the version. The versions are ordered by $wkey$; that is, $ov_1 < ov_2$ iff $ov_1.wkey < ov_2.wkey$. When a transaction reads the object, it gets the value of the highest version that is less than or equal to the transaction's key.

The keys in $[ov.wkey..ov.rkey]$ constitute the *interval* of ov, where $[i..j]$ denotes the set $\{i, i{+}1, \cdots, j\}$. While a version ov of an object exists, no transaction whose key is in $[ov.wkey..ov.rkey{-}1]$ can write into the object. This ensures that for any transaction that has read this version (such as the transaction using $ov.rkey$), ov continues to be the highest version less than or equal to the transaction's key. By not allowing such writes, the intervals of any two distinct versions ov_1 and ov_2 of an object have the following property: $[ov_1.wkey..ov_1.rkey{-}1] \cap [ov_2.wkey..ov_2.rkey{-}1] = \{\}$. Observe that $ov_1.rkey{=}ov_2.wkey$ holds iff a transaction with that key first read from ov_1 and then wrote into the object. Also observe that if a transaction writes a version of an object and a different transaction subsequently reads that version, then the first transaction cannot write into the object again.

State variables

In addition to the state variables H, *status*, and *allocated* of the database interface, we add the following:

aborted: powerset of KEYS. Initially empty.
Set of keys of aborted transactions.

done: powerset of KEYS. Initially empty.
Set of keys of transactions that have committed or aborted.

laststarted: KEYS. Initially 0.
Largest key allocated to a transaction.

versions(*obj*): powerset of VALUES×KEYS×KEYS.
Initially $\{ov : ov.value{=}\text{INITVALUE}(obj),\ ov.wkey{=}ov.rkey{=}0\}$.
Versions of *obj* currently maintained.

dependsupon(*key*): powerset of KEYS. Initially empty.

Set of keys that the transaction using *key* has read from; if $k \in dependsupon(key)$ then $k \neq key$ and *key* has read a version *ov* written by k.

S: sequence of $\{(id, Begin, key), (id, Read, key, obj, val), (id, Write, key, obj, val, \text{OK}), (id, End, key, \text{OK})\}$.
Initially, S is the null sequence.

An auxiliary variable. A serial history obtained by concatenating histories of the committed transactions in increasing order of their keys (timestamps).

The state variable H of the database interface becomes an auxiliary variable. We use $H(key)$ to denote the subsequence of H consisting of all entries using *key*. We use $S(<key)$ to denote the prefix of S consisting of all entries using keys less than *key*. Similarly, $S(>key)$ denotes the suffix of S consisting of all entries with keys higher than *key*. We continue to use our subscript notation to specify entries of a sequence. Thus, $S(>key)_i$ is the ith entry of $S(>key)$, and $S(>key)_{<i}$ is the prefix of $S(>key)$ consisting of all entries up to but excluding $S(>key)_i$.

Events

The following definition is to be used in the events that follow:

$$
\begin{aligned}
Abort(key) \equiv\ & aborted' = aborted \cup \{key\} \\
& \wedge\ done' = done \cup \{key\} \\
& \wedge\ [\forall obj: versions(obj)' = \{ov \in versions(obj): ov.wkey \neq key\}] \\
& \wedge\ status(id)' = \text{ABORTED} \\
& \wedge\ \neg allocated(key)' \\
& \wedge\ H' = H @ (id, Abort, key)
\end{aligned}
$$

Module implementation events are obtained by refining the database interface events. The notation <previous definition> refers to the event's definition given in Section 2.3.

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

$$
\begin{aligned}
Return(id, Begin, key) \equiv\ & \text{<previous definition>} \\
& \wedge\ key = laststarted' = laststarted + 1 \\
& \wedge\ dependsupon(key)' = \{\}
\end{aligned}
$$

$Return(id, Begin, \text{FAILED}) \equiv$ <previous definition>

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

$$
\begin{aligned}
Return(id, Read, key, obj, val) \equiv\ & \text{<previous definition>} \\
& \wedge\ dependsupon(key) \cap aborted = \{\} \\
& \wedge\ [\exists ov: ov = \max\{ov_1 \in versions(obj): ov_1.wkey \leq key\} \wedge val = ov.value \\
& \qquad \wedge\ ov.rkey' = \max(key, ov.rkey) \\
& \qquad \wedge\ dependsupon(key)' = dependsupon(key) \cup \{k: k = ov.wkey \wedge k \neq key\}]
\end{aligned}
$$

For $ov \in versions(obj)$, the notation $ov.rkey' = k$ means that $versions(obj)'$ is the same as $versions(obj)$ except that *ov* is updated as specified.

$Return(id, Read, key, obj, \text{ABORT}) \equiv$
$\quad status(id)=(Read, key, obj) \wedge dependsupon(key) \cap aborted \neq \{\}$
$\quad \wedge Abort(key)$

For the above to be a refinement of the corresponding database interface event, it is sufficient that $concurrentaccess(id)$ is true whenever $status(id)=(obj)$ and $dependsupon(key) \cap aborted \neq \{\}$ are true. The following safety requirement is to be proved:

$B_1 \equiv \quad keyof(id)=key \wedge status(id)=(obj) \wedge dependsupon(key) \cap aborted \neq \{\}$
$\quad \Rightarrow concurrentaccess(id)$

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

$Return(id, Write, key, obj, val, \text{OK}) \equiv$ <previous definition>
$\quad \wedge dependsupon(key) \cap aborted = \{\}$
$\quad \wedge \neg[\exists ov \in versions(obj)\text{: } key \in [ov.wkey..ov.rkey-1]]$
$\quad \wedge versions(obj)' = \{ov_1 \in versions(obj)\text{: } ov_1.wkey \neq key\}$
$\quad\quad \cup \{ov_2\text{: } ov_2.value = val \wedge ov_2.wkey = ov_2.rkey = key\}$

$Return(id, Write, key, obj, val, \text{ABORT}) \equiv$
$\quad status(id)=(Write, key, obj, val)$
$\quad \wedge [dependsupon(key) \cap aborted \neq \{\}$
$\quad\quad \vee [\exists ov \in versions(obj)\text{: } key \in [ov.wkey..ov.rkey-1]]]$
$\quad \wedge Abort(key)$

For the above to be a refinement of the corresponding database interface event, it is sufficient if the following is invariant in addition to B_1:

$B_2 \equiv \quad keyof(id)=key \wedge status(id)=(obj) \wedge ov \in versions(obj) \wedge key \in [ov.wkey..ov.rkey-1]$
$\quad \Rightarrow concurrentaccess(id)$

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

$Return(id, End, key, \text{OK}) \equiv$ <previous definition>
$\quad \wedge dependsupon(key) \subseteq done - aborted$
$\quad \wedge done' = done \cup \{key\}$
$\quad \wedge S' = S(<key)@H(key)@S(>key)$

$Return(id, End, key, \text{ABORT}) \equiv$
$\quad status(id)=(End, key) \wedge dependsupon(key) \cap aborted \neq \{\}$
$\quad \wedge Abort(key)$

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

$Return(id, Abort, key) \equiv$ <previous definition>
$\quad \wedge Abort(key)$

It can be easily checked that module implementation conditions **M2—M4** (in Section 2.5) are satisfied. Condition **M5** is established in the following two sections.

5.1. Proof of safety

We prove the following to be invariant, assuming that *correctkeyuse* is invariant:

$B_1 \equiv$ $keyof(id){=}key \wedge status(id){=}\langle obj\rangle \wedge dependsupon(key) \cap aborted \neq \{\}$
$\Rightarrow concurrentaccess(id)$

$B_2 \equiv$ $keyof(id){=}key \wedge status(id){=}\langle obj\rangle \wedge ov \in versions(obj) \wedge key \in [ov.wkey..ov.rkey-1]$
$\Rightarrow concurrentaccess(id)$

$B_3 \equiv$ $S_i{=}\langle Read, obj, val\rangle \Rightarrow val{=}lastvalue(obj, S_{<i})$.

B_1 and B_2 ensure that the implementation events are refinements of the database interface events. B_3 ensures serializability. We provide an informal justification of the invariance of B_1, B_2, and B_3. (Additional invariant assertions needed for a formal proof are indicated below.)

Assume that the antecedent of B_1 holds currently. Let $k_1 \in dependsupon(key) \cap aborted$, and let id_1 be the transaction that was allocated k_1. The key k_1 entered *dependsupon(key)* due to an occurrence of $Return(id, Read, key)$, which read from a version *ov* with $ov.wkey{=}k_1$. Clearly, transaction *id* was active and accessing *obj* when this event occurred. Transaction id_1 had accessed *obj* and was either active or committed when the event occurred, because *ov* existed. It could not be committed because k_1 is in *aborted.* Consequently, both *id* and id_1 were active when the *Read* returned and both had accessed *obj*. Hence *concurrentaccess* (*id*) was true, and continues to be true (by its definition).

Assume that the antecedent of B_2 holds currently. Let $ov.rkey{=}k_1$ and let id_1 be the transaction that was allocated k_1. The value of *ov.rkey* was set to k_1 due to an occurrence of $Return(id_1, Read, k_1)$, which read from *ov*. Clearly, transaction id_1 was active and accessing *obj* when this event occurred. Transaction *id* is currently active and accessing *obj*, because $status(id){=}\langle obj\rangle$. It suffices to show that transaction *id* was also active when the *Read* returned. This is true because from $key < k_1$, transaction *id* was active before transaction id_1 became active. Consequently, both *id* and id_1 were active when the *Read* returned, and both have accessed *obj*. Hence *concurrentaccess* (*id*) was true, and continues to be true (by its definition).

Consider B_3. The only event to affect B_3 is the $Return(End, key, \text{OK})$ event, which inserts $H(key)$ between $S(<key)$ and $S(>key)$. Thus B_3 is invariant if B_4, B_5 and B_6 shown below, are invariant:

$B_4 \equiv$ $key \notin done \wedge dependsupon(key) \subseteq done - aborted$
$\wedge H(key)_i{=}\langle Read, obj, val\rangle \wedge \langle obj\rangle \notin H(key)_{<i} \Rightarrow val{=}lastvalue(obj, S(<key))$

$B_5 \equiv$ $key \notin done \wedge dependsupon(key) \subseteq done - aborted$
$\wedge H(key)_i{=}\langle Read, obj, val\rangle \wedge \langle obj\rangle \in H(key)_{<i} \Rightarrow val{=}lastvalue(obj, H(key)_{<i})$

$B_6 \equiv$ $key \notin done \wedge dependsupon(key) \subseteq done - aborted$
$\wedge S(>key)_i{=}\langle Read, obj, val\rangle \wedge \langle obj\rangle \notin S(>key)_{<i} \Rightarrow \langle Write, obj\rangle \notin H(key)$

B_4 specifies that if the transaction using *key* can commit successfully and its first access to an object is a *Read*, then the value read is the last value in $S(<key)$. B_5 specifies that if a transaction can commit successfully and has read an object that was previously accessed by it, then the value read is the same as what was read or written in its previous access. B_6 specifies that if the transaction using *key* can commit successfully, and there are committed keys k_1 and k_2 such that $k_1 < key < k_2$ and k_2 has read a version written by k_1, then the transaction has not written the object. Therefore, the value read by k_2 will still be equal to the last value in $S(<k_2)$ after $H(key)$ is inserted into S.

To establish that B_4, B_5 and B_6 are invariant, we need to relate the versions of an object with its last values in $S(<key)$ and $H(key)$. The following invariant assertions relate these various values during the execution of the transaction using *key*:

$B_7 \equiv$ $key \notin done \wedge dependsupon(key) \cap aborted = \{\} \wedge H(key)_i = (Read, obj, val)$
$\wedge (obj) \notin H(key)_{<i} \Rightarrow$
$[\exists ov \in versions(obj): key \in [ov.wkey+1..ov.rkey] \wedge ov.value = val$
$\wedge ov.wkey \in dependsupon(key)]$

$B_8 \equiv$ $key \notin done \wedge dependsupon(key) \cap aborted = \{\} \wedge (obj) \in H(key) \Rightarrow$
$[\exists ov \in versions(obj): key \in [ov.wkey..ov.rkey] \wedge ov.value = lastvalue(obj, H(key))]$

$B_9 \equiv$ $S_i = (Read, k, obj, val) \wedge (k, obj) \notin S_{<i} \Rightarrow$
$[\exists ov \in versions(obj): k \in [ov.wkey+1..ov.rkey] \wedge ov.wkey \in done - aborted]$

$B_{10} \equiv$ $ov_1, ov_2 \in versions(obj) \wedge ov_1 \neq ov_2 \Rightarrow$
$[ov_1.wkey..ov_1.rkey-1] \cap [ov_1.wkey..ov_2.rkey-1] = \{\}$

$B_{11} \equiv$ $[\exists ov \in versions(obj): ov.value = val \wedge ov.wkey = key] \Leftrightarrow$
$[\exists (Write, key, obj, val) \in H: key \notin aborted]$

$B_{12} \equiv$ $[\exists ov \in versions(obj): ov.value = val \wedge ov.wkey = key \in done - aborted] \Leftrightarrow$
$[\exists (Write, key, obj, val) \in S]$

B_7 states that if the transaction using *key* is active and not about to be aborted, and its first access to an object was a read, then the version *ov* from which it read still exists and *ov.wkey* belongs to *dependsupon(key)*. B_8 states that if the transaction using *key* is active and not about to be aborted, and has accessed an object, then there is a version *ov* whose interval includes *key* and whose value equals the last value of the object in $H(key)$. B_9 states that if a committed transaction's first access to an object was a *Read*, then the version *ov* from which it read still exists and the transaction that wrote the version has committed. B_{10} through B_{12} state obvious properties.

We prove that B_4 is invariant given that B_7, B_{10}, B_{11} and B_{12} are invariant. Assume the antecedent of B_4. The antecedent of B_7 is satisfied. Hence, there exists $ov \in versions(obj)$ such that $ov.value = val$, $key \in [ov.wkey+1..ov.rkey]$, and $ov.wkey \in dependsupon(key)$. This last condition and the antecedent of B_4 together imply that the transaction allocated *ov.wkey* has committed. Thus, S contains the entry $(Write, ov.wkey, obj, val)$, by B_{12}. The existence of *ov* also implies that H, and hence S, does not contain an entry $(Write, k, obj)$ with $k \in [ov.wkey..ov.rkey-1]$, by B_{10} and B_{11}. Therefore, $val = lastvalue(obj, S(<key))$, which is the consequent of B_4.

We prove that B_5 is invariant given that B_8 and B_{10} are invariant. B_5 holds initially. It is preserved by every event occurrence. The only nontrivial case is an occurrence of $Return(Read, key, obj, val)$. Assume the antecedent of B_8, which is implied by the antecedent of B_5. From the consequent of B_8 and from B_{10}, we see that the value returned by the *Read* is $lastvalue(obj, H(key)_{<i})$.

We prove that B_6 is invariant given that B_9, B_{10} and B_{11} are invariant. Assume the antecedent of B_6, namely: for some key k and some i, $S(>key)_i = (Read, k, obj, val)$ and $(obj) \notin S(>key)_{<i}$. From B_9, there is an $ov \in versions(obj)$ such that $k \in [ov.wkey+1..ov.rkey]$ and *ov.wkey* is committed. Because $(obj) \notin S(>key)_{<i}$ and *key* is not committed, it follows that $ov.wkey < key$. Because $(key) \notin S(>key)$, we have $k > key$. Thus, $key \in [ov.wkey+1..ov.rkey-1]$ and transaction *id* could not have written *obj*, by B_{10} and B_{11}.

We still have to establish that B_7 through B_{12} are invariant. B_7 and B_8 hold initially, because $(obj) \notin H(key)$. Successful reads and writes preserve B_7 and B_8. A version ov referred to in their consequents ceases to exist only if the transaction using *ov.wkey* aborts, in which case $dependsupon(key) \cap aborted$ is not empty and B_7 and B_8 hold vacuously.

B_9 holds initially because S is the null sequence. B_9 is affected only by a transaction committing, when $H(key)$ is inserted into S. B_9 is preserved because of B_7, and because *key* is committed only after all the keys it depends upon have committed.

B_{10} through B_{12} hold initially. It is easy to see that they are preserved by every event occurrence.

Lemma 6. $B_{1-2} \wedge B_{7-12}$ are invariant, assuming that *correctkeyuse* is invariant. (Proof omitted.)

To prove the above lemma formally, by showing that the assertions satisfy the invariance rule, additional assertions are needed. Let us retrace the steps our proof of safetly. Recall that serializability is ensured by the invariance of B_3. The invariance of B_{7-12} is sufficient for the invariance of B_{4-6}, which is sufficient for the invariance of B_3.

5.2. Proof of progress

We prove that the multi-version timestamp module implementation satisfies the progress requirement in the database interface specification, assuming that the following fairness requirement is satisfied:

F1: Every event that is not a call event has weak fairness.

In the following, we use *lastdone* to denote the largest key such that $[0..lastdone] \subseteq done$. We use $lastdone \geq dependsupon(key)$ to mean $lastdone \geq k$ for every $k \in dependsupon(key)$.

Lemma 7. The following progress assertions hold:

$X_1 \equiv$ $status(id)=(Begin)$ *leads-to* $status(id) \in \{\text{READY, NOTBEGUN}\}$

$X_2 \equiv$ $status(id)=(Abort)$ *leads-to* $status(id)=\text{ABORTED}$

$X_3 \equiv$ $status(id)=(obj)$ *leads-to* $status(id) \in \{\text{READY, ABORTED}\}$

$X_4 \equiv$ $status(id)=(End, key) \wedge lastdone \geq dependsupon(key)$ *leads-to* $status(id) \in \{\text{COMMITTED, ABORTED}\}$

$X_5 \equiv$ $lastdone=j \wedge laststarted>j$ *leads-to* $lastdone>j$

Proof.
X_1 and X_2 are proved exactly as W_1 and W_3 are proved for the two-phase locking implementation (in proof of Lemma 2).

X_3 holds as follows. Assume $status(id)=(Read, key, obj)$. If $dependsupon(key) \cap aborted = \{\}$, then $Return(Read, key, val)$ is continuously enabled; it eventually occurs, resulting in $status(id)=\text{READY}$, unless $dependsupon(key) \cap aborted$ becomes nonempty. If the latter happens, then $Return(Read, key, \text{ABORT})$ is continuously enabled and it eventually occurs, resulting in $status(id)=\text{ABORTED}$. The argument for $status(id)=(Write, key, obj)$ is similar.

X_4 holds as follows. Assume $status(id)=(End, key)$ and $lastdone \geq dependsupon(key)$. One of either $Return(End, key, \text{ABORT})$ or $Return(End, key, \text{OK})$ is continuously enabled and it eventually occurs. Occurrence of the former results in $status(id)=\text{ABORTED}$, the latter in $status(id)=\text{COMMITTED}$.

X_5 holds as follows. Assume $lastdone=j \wedge laststarted>j$. Thus, all transactions with keys less than or

equal to j have either committed or aborted. Consider the transaction using key $j+1$. This transaction is active, otherwise *lastdone* would be greater than j. From L_2, it eventually issues a *Call*(*End*), unless it is aborted (during a write attempt or due to an abort request). If it is aborted, then *lastdone* increases. Assume that the transaction issues *Call*(*End*). Because all transactions it depends upon have committed (otherwise it would have aborted), it commits and *lastdone* increases. Thus, in each case, $lastdone > j$ holds.
End of proof.

Applying the transitivity rule repeatedly to X_5, we get $status(id)=(End, key)$ *leads-to* $lastdone \geq dependsupon(key)$. Combining this with X_4, we get $status(id)=(End)$ *leads-to* $status(id) \in \{$COMMITTED, ABORTED$\}$. Applying the disjunction rule to this and X_1 through X_3, we get the desired progress property.

6. Implementation of Procedure Calls

Lamport's informal specification of a module interface consists of a set of procedures [12]. In our model, each interface procedure P is represented by *Call*(id, P) and *Return*(id, P) events. In a practical programming language, such as Pascal or C, the return of a procedure call transfers control and parameter values only. State variables are updated during the call execution by atomic events that constitute the body of the procedure. In our specification of implementation events, however, *nonauxiliary* state variables can also be updated as part of the atomic action of *Return*(id, P). For example, given an interface procedure P with input parameters **x** and result parameters **y**, we have implementation events of the following form:

$$\begin{aligned} Return(id, P, \mathbf{x}, \mathbf{y}) \equiv\ & status(id)=(P, \mathbf{x}) \\ & \wedge status(id)'=\text{READY} \ \wedge\ \mathbf{y}=f(\mathbf{v}) \ \wedge\ \mathbf{v}'=g(\mathbf{v}) \end{aligned}$$

where, f and g are functions and **v** is a subset of state variables, some of which are nonauxiliary. The second and third conjuncts in the above event formula represent the transfer of control and parameter values respectively, and the last conjunct specifies the update of state variables.

To satisfy the practical requirement that the return of a procedure call transfers only control and parameter values, the implementation specifications in this paper need to be refined further. We briefly discuss how such a refinement can be carried out without actually doing it.

Let all the variables in **v** become auxiliary variables. Introduce additional state variables **u**. Let **w** be a subset of **u** that holds the result parameters. The return event above is refined to have the following form:

$$\begin{aligned} Return(id, P, \mathbf{x}, \mathbf{y}) \equiv\ & status(id)=(P, \mathbf{x}) \wedge finished(\mathbf{u}) \\ & \wedge status(id)'=\text{READY} \ \wedge\ \mathbf{y}=\mathbf{w} \ \wedge\ \mathbf{v}'=g(\mathbf{v}) \end{aligned}$$

where *finished* is a boolean function of **u**. Note that aside from $status(id)$, the state variables that are updated in the action of the above event are auxiliary. Hence, it satisfies the requirement of a practical programming language stated above. For this new event to be a refinement of the old event, however, we will have to establish the following to be invariant:

$$status(id)=(P, \mathbf{x}) \wedge finished(\mathbf{u}) \Rightarrow \mathbf{w}=f(\mathbf{v})$$

To update the state variables in **u**, we need to introduce a set of events $\{body_i, i=1, \cdots, n\}$ that constitute the body of the procedure P. Each such event has the following form:

$$body_i(id, P, \mathbf{x}) \equiv status(id)=(P, \mathbf{x}) \wedge b_i(\mathbf{u}) \\ \wedge \mathbf{u}'=h_i(\mathbf{u})$$

where b_i is a boolean function of $\mathbf{u}$ and $\mathbf{u}'=h_i(\mathbf{u})$ represents a computation that the new implementation can perform as an atomic action. Observe that each $body_i$ event is a refinement because it has a null image with respect to $\mathbf{v}$, the state variables of the old implementation. These events perform the update specified by the function g in the old event, but in n atomic actions instead of one.

The above approach is similar to one suggested by Lamport [9], where he transforms the nonauxiliary state variables in $\mathbf{v}$ in the old implementation to auxiliary state functions in the new implementation.

7. Conclusions

We have presented a formalism for specifying interfaces and implementations of program modules in a hierarchy. Our formalism is based upon the relational notation for specifying state transition systems. A module interface is specified by a state transition system and a set of invariant and progress requirements. A module implementation is specified by a state transition system and some fairness requirements for events. A module offers an upper interface to a user and it may use one or more lower interfaces each offered by a module at a lower level of the hierarchy. In Section 2.5, we give some sufficient conditions for a module implementation to satisfy its interfaces. They are referred to as module implementation conditions **M1—M5**.

In Sections 3-5, we specify a serializable database interface and two module implementations that offer the interface. For each implementation, we also sketch a proof that it satisfies the serializable database interface.

For a module A, the notion "*A offers B*" where B is an interface, is just a specialization of the notion "*A implements B*" where B is a service specification. The meaning of *A implements B* in our formalism is essentially the same as that in other state-transition formalisms [4,10,11,13]. The basic definition of *A implements B* in each formalism is the following: every "observable behavior" of A is an "allowed behavior" of B. Our formalization of the more general notion "*A uses C to offer B*" appears to be new.

While it is conceptually simple to define *A implements B* in terms of behaviors, some convenient way of specifying the sets of observable and allowed behaviors and reasoning with them is needed in practice. Specifically, it is desirable to have conditions that require reasoning with just the states and state transitions of A and B, instead of sets of behaviors. Towards this end, Lamport defined a *refinement* mapping from A to B [1,10], Lynch and Tuttle defined a *possibilities* mapping from A to B [13], and we defined a *projection* mapping from A to B [6,8,21].

Refinement and projection mappings are conceptually the same. It might appear that these mappings are less general than a multi-valued possibilities mapping. This is not so, however, because the extra information in a multi-valued possibilities mapping can be provided by auxiliary variables and invariant requirements; see Section 2.3 and [8].

The module implementation conditions that guarantee *A uses C to offer B* are based upon the refinement relation between state transition systems, which was adapted to the relational notation [8] from projection mapping [6,21].

In the context of *A offers B*, where B is a module interface, we found it convenient to define an observable behavior of A to be a sequence of observable events; this is the approach of Lynch and Tuttle [13]. (Such an event sequence, referred to in the literature as a *trace*, is used exclusively in process-based approaches such as CSP [5].) In most state-transition formalisms [1,4,6], the observable behavior of a module is a sequence of states. There is, however, no fundamental difference in the two approaches. By including an auxiliary variable, the observable state of the module can be defined to include the sequence of observable events. For example, an auxiliary variable T can be included in the module, and the conjunct $T'=T@image(f)$ is added to every event f of the module, where $image(f)$ is the "observable image" of f on the interface.

In formulating module implementation conditions, there are some advantages to having events observable on an interface. First, with a set of interface events, we can give a precise description of how two modules that share an interface interact; specifically, by dividing interface events into input events and output events, we can specify for each interface event the module that controls it. (This type of information is very important for implementors.) A second advantage of having events is that it is easy to eliminate a vacuous implementation; for example, our condition **M4** suffices.

The various formalisms differ greatly in how they allow modules to be composed. The approach of Lynch and Tuttle [13], as well as some temporal-logic approaches [17,20], allow the composition of arbitrary networks of modules. Because of the generality, applications of these approaches require a great deal of notation.

We restrict the network structure to be a tree, with each tree node being a program module. Specifically, we impose a hierarchical relationship between two modules that share an interface, such that one module is a user and the other is the service provider. Note that this hierarchical relationship is encountered in many large software systems, in addition to the examples of this paper, e.g., communication networks. By sacrificing some generality, we get fairly simple conditions for module compositon (i.e., the module implementation conditions). Since each module in our model can be a network of processes, our model is still applicable to the specification of many software systems.

A consequence of the hierarchical structure is the following: If a module uses a lower interface offered by another module, the lower interface events are invisible on the upper interface; thus, in our model, there is no need for renaming of events [13]. There are a couple of differences between our approach and the approach of Lynch and Tuttle [13] that are not the result of the hierarchical structure. First, we do not require an externally-controlled event e to be always enabled. It is sufficient that e is enabled whenever the interface is allowed to initiate it; thus in formulating **M4**, we make use of the invariant requirements in an interface specification. Second, we allow a module to have events that do not have fairness requirements; that is, only some events controlled by a module need to be fairly scheduled in an actual implementation. These fairness requirements are explicitly stated and are part of the module specification.

The approaches of Chandy and Misra [4] and Lamport [11] are also top-down. That is, starting with an interface specification, they derive, by stepwise refinement, an implementation that offers the interface. However, the notion of a lower interface that can be used by the implementation is not considered in their approaches.

Lynch et al. have applied the I/O automaton model to specify transaction-processing systems [14]. Even though their work and ours were parallel developments, the two approaches towards the modeling of database systems have a number of similarities. This may be because both approaches are based upon state-transition formalisms.

In another paper [22], we have developed a stepwise refinement heuristic by extending the refinement relation between state transition systems to a *conditional refinement* relation. The specification of transport-layer communication protocols that use various lower interfaces is considered.

Each lower interface represents one type of service provided by the network layer of a communication network. Specifically, several sliding window protocols that provide reliable data transfer for network layers that can lose, duplicate and reorder messages are presented. In [16], the specification of a complete transport protocol, with the functions of connection management and reliable data transfer in both directions, is considered. The transport protocol is derived by composing single-function protocols such that it is the refinement of each instance of the single-function protocols.

All of the state-transition and temporal-logic formalisms referenced above use the interleaving model of concurrent execution. The interleaving model is fundamentally different from process-based models, such as CSP [5]. Process-based models generally use traces for reasoning and allow the composition of arbitrary networks of modules. The absence of a global state, however, makes it difficult to relate a module implementation to its interface specifications [2,5].

Lastly, some readers might object to our use of a state transition system for specifying an interface. In our experience, such an approach is crucial to the specification of nontrivial systems. Conceptually, the use of a state transition system to specify an interface does not really constrain an implementor, for the following reason: in **M1**, we require that a user of the interface can access the state variables of the interface only through the interface events. Thus, all of the interface state variables can be made into auxiliary variables in a module that offers the interface. For example, the history variable H of the database interface in Section 3 is made into an an auxiliary variable in both the two-phase locking and multi-version timestamp implementations. In practice, however, the state transition system specifying an interface will bias implementors of modules that offer the interface; in fact, the larger the state transition system, the greater will be the bias. We believe that implementation-independence is not in itself a desirable attribute of specifications. A specification should be of help to an implementor, guiding him but without seriously limiting his options. We believe that our formalism is flexible enough for expressing specifications with the appropriate balance.

Acknowledgements: We thank the anonymous referees for their constructive criticisms. The presentation of this paper is much improved because of their diligence. We also thank Ken Calvert of the University of Texas at Austin for his helpful comments.

References

[1] M. Abadi and L. Lamport, *The existence of refinement mappings*, Technical Report, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, August 1988.

[2] R.J.R. Back and R. Kurki-Suonio, ''Distributed cooperation with action systems,'' *ACM TOPLAS*, Vol. 10, No. 4, October 1988, pp. 513-554.

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

[4] K.M. Chandy and J. Misra, *A Foundation of Parallel Program Design*, Addison-Wesley, Reading, Massachusetts, 1988.

[5] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.

[6] S.S. Lam and A.U. Shankar, "Protocol verification via projections," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 10, July 1984, pp. 325-342.

[7] S.S. Lam and A.U. Shankar, "Specifying an implementation to satisfy interface specifications: A state transition approach," 26th Lake Arrowhead Workshop on *How will we specify concurrent systems in the year 2000?*, September 1987.

[8] S.S. Lam and A.U. Shankar, "A relational notation for state transition systems," invited talk at the Eighth International Symposium on Protocol Specification, Testing, and Verification, Atlantic City, N.J., June 1988; full version available as Technical Report TR-88-21, Department of Computer Sciences, University of Texas at Austin, May 1988 (revised August 1989).

[9] L. Lamport, "An assertional correctness proof of a distributed algorithm," *Science of Computer Programming*, Vol. 2, 1982, pp. 175-206.

[10] L. Lamport, "Specifying concurrent program modules," *ACM TOPLAS*, Vol. 5, No. 2, April 1983, pp. 190-222.

[11] L. Lamport, "What it means for a concurrent program to satisfy a specification: Why no one has specified priority," *Proc. 12th ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985.

[12] L. Lamport, "A serializable database interface," 26th Lake Arrowhead Workshop on *How will we specify concurrent systems in the year 2000?*, September 1987.

[13] N.A. Lynch and M.R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., August 1987.

[14] N. Lynch, M. Merritt, W. Weihl and A. Fekete, "A Theory of Atomic Transactions," Technical Report MIT/LCS/TM-362, Laboratory for Computer Science, M.I.T, June 1988.

[15] Z. Manna and A. Pnueli, "Adequate proof principles for invariance and liveness properties of concurrent programs," *Science of Computer Programming*, Vol. 4, 1984.

[16] S.L. Murphy and A.U. Shankar, "Service specification and protocol construction for the transport layer," CS-TR-2033, UMIACS-TR-88-38, Computer Science Dept., Univ. of Maryland, May 1988; an abbreviated version appears in *Proc. ACM SIGCOMM '88 Symposium*, August 1988.

[17] V. Nguyen, A. Demers, D. Gries, S. Owicki, "A model and temporal proof system for networks of processes," *Distributed Computing*, Vol. 1, No. 1, 1986.

[18] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Informatica*, Vol. 6, 1976, pp. 319-340.

[19] S. Owicki and L. Lamport, "Proving liveness properties of concurrent systems," *ACM TOPLAS*, Vol. 4, No. 3, 1982.

[20] A. Pnueli, "In transition from global to modular temporal reasoning about programs," NATO ASI Series, Vol. F13, *Logics and Models of Concurrent Systems*, K.R. Apt (ed.), Springer-Verlag, Berlin, Heidelberg, 1985, pp. 123-144.

[21] A.U. Shankar and S.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.

[22] A.U. Shankar and S.S. Lam, "Time-dependent distributed systems: Proving safety, liveness and real-time properties," *Distributed Computing*, Vol. 2, No. 2, 1987.

[23] A.U. Shankar and S.S. Lam, "A stepwise refinement heuristic for protocol construction," Technical report UMIACS-TR-87-12, University of Maryland, College Park, March 1987 (revised March 1989).

Appendix A

A proof of Lemma 1 is given in two steps. First, we prove the invariance of the following formulas, which specify that every allocated key is associated with a unique active transaction:

$A_8 \equiv \quad \neg allocated(key) \Rightarrow [\forall id: keyof(id) \neq key]$

$A_9 \equiv \quad allocated(key) \Rightarrow [\exists \text{exactly one } id: keyof(id)=key]$

Lemma A1. $A_8 \wedge A_9$ satisfies the invariance rule, assuming that *correctkeyuse* is invariant. (Proof omitted.)

To prove Lemma 1, namely, $A_1 \wedge A_{4-7}$ is invariant, we need the following formulas, which specify relationships between state variables during the growing stage of a transaction. During this stage, the transaction acquires a key and then acquires locks.

$A_{10} \equiv \quad status(id) \in \{\text{NOTBEGUN}, (Begin)\} \Rightarrow (id) \notin H$

$A_{11} \equiv \quad keyof(id)=key \wedge status(id)=\text{READY} \Rightarrow status_L(key)=\text{READY}$

$A_5 \equiv \quad keyof(id)=key \wedge \neg locked(key, obj) \Rightarrow (id, obj) \notin H \wedge localvalue(obj, key)=\text{NULL}$

$A_{12} \equiv \quad keyof(id)=key \wedge status_L(key)=(AcqLock, obj) \Rightarrow \neg locked(key, obj) \wedge status(id)=(key, obj)$

$A_6 \equiv \quad keyof(id)=key \wedge locked(key, obj) \wedge status(id) \neq (End) \Rightarrow$
$\quad storedvalue(obj)=lastvalue(obj, S) \wedge$
(a) $\quad [((id, obj) \notin H \wedge localvalue(obj, key)=\text{NULL}) \vee$
(b) $\quad ((id, obj) \notin H \wedge localvalue(obj, key)=lastvalue(obj, S)) \vee$
(c) $\quad ((id, obj) \in H \wedge localvalue(obj, key)=lastvalue(obj, H(id)))]$

$A_{13} \equiv \quad keyof(id)=key \wedge status(id)=(key, obj) \wedge locked(key, obj) \wedge localvalue(obj, key)=\text{NULL}$
$\quad \Rightarrow (id, obj) \notin H$

$A_{14} \equiv \quad keyof(id)=key \wedge status_L(key)=(Read_L, obj) \Rightarrow locked(key, obj)$
$\quad \wedge localvalue(obj, key)=\text{NULL} \wedge (id, obj) \notin H \wedge status(id)=(Read, key, obj)$

$A_{15} \equiv \quad keyof(id)=key \wedge locked(key, obj) \Rightarrow obj \in accessed(id)$

The following formulas specify relationships when a transaction is aborting:

$A_{16} \equiv \quad keyof(id)=key \wedge aborting(key) \Rightarrow [\exists obj: status(id)=(key, obj)]$

$A_1 \equiv \quad keyof(id)=key \wedge aborting(key) \Rightarrow concurrentaccess(id)$

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

$A_{17} \equiv \quad keyof(id)=key \wedge locked(key, obj) \wedge status(id)=(End) \wedge localvalue(obj, key) \neq \text{NULL}$
$\quad \Rightarrow storedvalue(obj)=lastvalue(obj, S)$

$A_{18} \equiv \quad keyof(id)=key \wedge status_L(key)=(Write_L, obj, val)$
$\quad \Rightarrow status(id)=(End, key) \wedge val=lastvalue(obj, H(id)) \wedge locked(key, obj)$

$A_7 \equiv \quad keyof(id)=key \wedge locked(key, obj) \wedge status(id)=(End) \Rightarrow$
$\quad (id, obj) \in H \wedge$
(a) $\quad [(localvalue(obj, key)=lastvalue(obj, H(id)) \wedge storedvalue(obj)=lastvalue(obj, S))$

(b) $\vee$ ($localvalue\,(obj, key)$=NULL $\wedge$ $storedvalue\,(obj)$=$lastvalue\,(obj, H\,(id)))]$

The following formulas specify relationships during the lock-releasing stage of a transaction:

$A_{19} \equiv$ $\neg allocated\,(key) \Rightarrow localvalue\,(obj, key)$=NULL

$A_{20} \equiv$ $status_L\,(key)$=$(RelLock\,, obj) \Rightarrow \neg allocated\,(key) \wedge locked\,(key\,, obj)$

$A_{21} \equiv$ $\neg locked\,(key\,, obj) \Rightarrow localvalue\,(obj, key)$=NULL

The following are also needed:

$A_4 \equiv$ $(\forall key: \neg locked\,(key\,, obj)) \Rightarrow storedvalue\,(obj)$=$lastvalue\,(obj, S\,)$

$A_{22} \equiv$ $owned\,(key\,, obj) \Leftrightarrow locked\,(key\,, obj)$

$A_{23} \equiv$ $owned\,(key\,, obj) \Rightarrow (\forall k \neq key: \neg owned\,(k\,, obj))$.

Lemma A2. $A_1 \wedge A_{4\text{–}7} \wedge A_{10\text{–}23}$ satisfies the invariance rule, given that $A_8 \wedge A_9$ is invariant. (Proof omitted.)