Distrib Comput (1992) 6:39–63

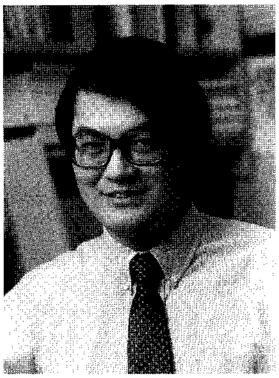
DISTRIBUTED COMPUTING

Specifying modules to satisfy interfaces: a state transition system approach*

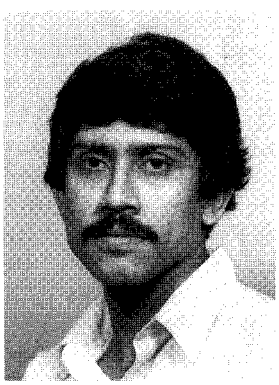
Simon S. Lam and A. Udaya Shankar

[1] Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA
[2] Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

Received August 1988 / Accepted August 1991



Simon S. Lam is Chairman of the Department of Computer Sciences at the University of Texas at Austin and holds an endowed professorship. His research interests are in the areas of computer networks, communication protocols, performance models, formal methods, and network security. He serves on the editorial boards of *IEEE Transactions on Software Engineering* and *Performance Evaluation*. He is an IEEE Fellow, and was a corecipient of the 1975 Leonard G. Abraham Prize Paper Award from the IEEE Communications Society. He organized and was program chairman of the first ACM SIGCOMM Symposium on Communications Architectures and Protocols in 1983. He received the BSEE degree (with Distinction) from Washington State University in 1969, and the MS and Ph.D. degrees from the University of California at Los Angeles in 1970 and 1974 respectively. Prior to joining the University of Texas faculty, he was with the IBM T.J. Watson Research Center from 1974 to 1977.



A. Udaya Shankar received the B. Tech. degree in Electrical Engineering from the Indian Institute of Technology, Kanpur, in 1976, the M.S. degree in Computer Engineering from Syracuse University, Syracuse, NY, in 1978, and the Ph.D. degree in Electrical Engineering from the University of Texas at Austin, in 1982. Since January 1983, he has been with the University of Maryland, College Park, where he is now an Associate Professor of Computer Science. Since September 1985, he has been with the Institute for Advanced Computer Studies at the University of Maryland. His current research interests include the modeling and analysis of distributed systems and network protocols, from both correctness and performance aspects. He is a member of IEEE and ACM.

Summary. We define *interface*, *module* and the meaning of *M offers I*, where *M* denotes a module and *I* an interface. For a module *M* and disjoint interfaces *U* and *L*, the meaning of *M using L offers U* is also defined. For a linear hierarchy of modules and interfaces, M_1, I_1, $M_2, I_2, \ldots, M_n, I_n$, we present the following composition theorem: If M_1 offers I_1 and, for $i=2, \ldots, n$, M_i using I_{i-1} offers I_i, then the hierarchy of modules offers I_n.

Our theory is applied to solve a problem posed by Leslie Lamport at the 1987 Lake Arrowhead Workshop. We first present a formal specification of a serializable database interface. We then provide specifications of two modules, one based upon two-phase locking and the other multi-version timestamps; the two-phase locking module uses an interface offered by a physical database. We prove that each module offers the serializable interface.

Key words: Interface – Module – Specification – Verification – Composition

1 Introduction

Consider a *module* that provides services to a user. Interactions between the module and user take place at an *interface*. In our theory, an interface is specified by a set of allowed sequences of interface events; each such sequence defines an allowed sequence of interactions between the module and user. For a module *M* and an interface *I*, we define the meaning of *M offers I* (see Sect. 2). Our definition is similar to–but not quite the same as–various definitions of *M satisfies S* in the literature, where *S* is a specification of *M* [1, 4–7, 9, 12, 14, 15]. Most definitions of *M satisfies S* have this informal

* The work of Simon S. Lam was supported by National Science Foundation grants no. NCR-8613338 and no. NCR-9004464. The work of A. Udaya Shankar was supported by National Science Foundation grants no. ECS-8502113 and no. NCR-8904590

Correspondence to: S.S.Lam

meaning: M *satisfies* S if every possible observation of M is described by S. Specific definitions, however, differ in whether interface events or states are observable, in whether observations are finite or infinite sequences, as well as in the particular formalism for representing these sequences.

Differences also arise because the method of interaction at an interface is different in different models. Let us consider models in which module observations are interface event sequences [5, 14, 15]. We identify three requirements that characterize interactions at an interface. First, the occurrence of an interface event requires *simultaneous participation* by a module and its environment; moreover, such occurrence is observable by both the module and its environment. This requirement appears to be fundamental and is included in all models that we are familiar with.

The second requirement, which we call *unilateral control*, specifies that each interface event is under the control of either the module or its environment. Specifically, the set of interface events is partitioned into a set of *input events* controlled by the environment and a set of *output events* controlled by the module. The side (module or environment) with control of an interface event is the only one that can initiate the event's occurrence. This notion of unilateral control is used in the I/O automata model of Lynch and Tuttle [15], and also described by Lamport [14].

Since the occurrence of an interface event requires simultaneous participation by both sides of the interface, it is possible that an interface event initiated by one side cannot occur because the other side refuses to participate. In the model of I/O automata [15, 16], such possibility is eliminated by a third requirement: each I/O automaton is *input-enabled*, i.e., every input event is enabled in every state of the automaton. With this requirement, the class of interface specifications becomes somewhat restricted; for example, a module with a finite input buffer such that inputs causing overflow are blocked cannot be specified.

In our theory, interface interactions are characterized by both the requirements of simultaneous participation and unilateral control. However, a module is required to be input-enabled *only when* the occurrence of an input event would be safe (this notion will be formally defined). For an input event whose occurrence would be unsafe, the module has a choice: it may let the event occur or it may block (disable) the event's occurrence. For example, blocking is useful for the specification of many communication protocols that enforce input control, flow control or congestion control.

Two modules interacting across an interface can be composed to become a single module by hiding the interface between them. In this respect, the composition of two modules in our theory is defined in a manner not unlike the approaches of CSP [5] and I/O automata [15]. However, in developing our theory, our vision of how it should be applied is different from those in [5, 15]; specifically, we are more interested in decomposing the specification of a complex system than in composition per se. An elaboration on this point follows.

Suppose an interface I has been specified through which a system provides services. Instead of designing and implementing a monolithic module M that offers I, we would like to implement the system as a collection of smaller modules $\{M_i\}$ such that the composition of $\{M_i\}$ offers I. To achieve this objective, the following three-step approach may be used:

Step 1. Derive a set of interfaces $\{S_i\}$ from I, one for each module in the collection (*decomposition* step).

Step 2. Design modules individually and, for all i, prove that M_i offers S_i *assuming* that the environment of M_i satisfies S_i in some manner.

Step 3. Apply an inference rule (*composition theorem*) to infer from the proofs in Step 2 that the composition of $\{M_i\}$ offers I.

The above approach has the following highly-desirable feature: given interfaces $\{S_i\}$, each module can be designed and implemented individually. However, the decomposition step – i.e., deriving the interfaces $\{S_i\}$ from I – is not easy to do. (We will say more about this below.) Furthermore, to develop the approach into a valid method, the following problem has to be solved, namely: In general, the inference rule required in Step 3 uses circular reasoning, and may not be valid. To see this, consider modules M and N that interact across interface I. Each module guarantees some properties of I only if its environment satisfies certain properties of I. However, module M is part of the environment of module N, and module N is part of the environment of module M.

The above problem was considered by Misra and Chandy [18] for processes that communicate by CSP primitives. They gave a proof rule for assumptions and guarantees that are restricted to safety properties. Using different models, Pnueli [22] presented a proof rule and Abadi and Lamport [2] presented a composition principle that are more general than the rule of Misra and Chandy in that assertions of assumptions and guarantees can be progress properties (albeit the class of assertions is still restricted)..

In thinking about an interface, we depart from the usual notion that it is the "external view" of a particular module, with a separate one specified for each module. Instead, we think of an interface as being *two-sided*, namely: there is a service provider on one side of the interface, and a user on the other, with both the user's behaviors and the service provider's behaviors constrained by the same set of interface event sequences; in this respect, an interface is symmetric. However, in our definitions of M *offers* I and M *using* L *offers* U (see Sect. 2), the user and the service provider of each interface have asymmetric obligations. By organizing modules hierarchically and having asymmetric obligations for each interface, circular reasoning is avoided.

For example, consider module M in Fig. 1. It provides services to a user through interface U while it uses user

module M — upper interface U

module N — lower interface L

Fig. 1. Module M and its environment

services offered by another module through interface L. We refer to U as the *upper interface* and L as the *lower interface* of module M. Note that module M is the user of interface L and the service provider of interface U. Its environment consists of both the user of U and the module that offers L.

Many practical systems have a hierarchical structure. In fact, almost all computer networks have layered protocol architectures. Each protocol layer–e.g., transport, data link–corresponds to a module in our composition theorem. (Note that each protocol layer is composed of a set of entities [19, 23, 24]. We place no restriction on how these entities are composed.)

In [10], our theory and composition theorem have been extended to a general model of layered systems in which each layer is a set of modules; each module may offer multiple disjoint upper interfaces and use multiple disjoint lower interfaces. More precisely, a layered system in [10] is a directed acyclic graph where each node is a module, and each arc, say an arc from node M to node N, represents an interface whose service provider is N and whose user is M. (Note that the upper interface of any module can be made into a set of disjoint interfaces, one for a different user, by simply tagging interface events with user names.)

Organizing modules hierarchically has an additional benefit bacause interfaces are also organized hierarchically. Suppose interface I is the topmost interface offered to users of the hierarchy of modules. Other interfaces in the hierarchy can be derived from I by a topdown approach as follows. Consider some interface U in the hierarchy. To design a module M that offers U, we may assume that certain services are offered by other modules through a set of disjoint interfaces $\{L_j, j \in J\}$. In this manner, interfaces offered by other modules at lower levels of the hierarchy are derived and specified.

The balance this paper is organized as follows. In Sect. 2, we first present our theory in a general semantic framework, and then a specification formalism suitable for practical application. In Sections 3–6, we present our solution to a problem posed by Lamport [13]. Specifically, in Sect. 3, we present a specification of a serializable database interface, to be called upper interface U. In Sect. 4, we specify an interface for accessing a physical database, to be called lower interface L. In Sect. 5, a module based upon two-phase locking is specified, and a proof that it satisfies *M using L offers U* is given. In Sect. 6, a different module, based upon multi-version timestamps, is specified; a proof that it satisfies *M offers U* is given. In Sect. 7, we discuss how events in our notation can be further refined to satisfy atomicity requirements of a practical programming language. Some concluding remarks are given in Sect. 8.

2 Theory and notation

In Sects. 2.1 and 2.2, we define *interface, state transition system, module, M offers I*, and *M using L offers U*, where I, U and L are interfaces and M is a module. Our key result is a composition theorem for a linear hierarchy of modules and interfaces. In Sects. 2.3 and 2.4, our definitions and results are recast in the *relational notation* [9], which is used to specify database interfaces and modules in Sects. 3–6. In Sect. 2.5, we elaborate on how to use the relational notation.

2.1 *Interface, state transition system and module*

We first define some notation for sequences. A *sequence over E*, where E is a set, means a (finite or infinite) sequence $(e_0, e_1, \ldots)$, where $e_i \in E$ for all i. A *sequence over alternating E and F*, where E and F are sets, means a sequence $(e_0, f_0, e_1, f_1, \ldots)$, where $e_i \in E$ and $f_i \in F$ for all i.

Definition. An interface I is defined by:

- *Events*(I), a set of events that is the union of two disjoint sets,
  Inputs(I), a set of input events, and
  Outputs(I), a set of output events.
- *AllowedEventSeqs*(I), a set of sequences over *Events*(I), each of which is referred to as an allowed event sequence of I.

For a given interface I, define

SafeEventSeqs$(I) = \{w : w$ is a finite prefix of an allowed event sequence of $I\}$

which includes the empty sequence.

Definition. A state transition system A is defined by:

- *States*(A), a set of states.
- *Initial*(A), a subset of *States*(A), referred to as initial states.
- *Events*(A), a set of events.
- *Transitions*$_A(e)$, a subset of *States*$(A) \times$ *States*(A), for every $e \in$ *Events*(A). Each element of *Transitions*$_A(e)$ is an ordered pair of states referred to as a transition of e.

A *behavior* of A is a sequence $\sigma = (s_0, e_0, s_1, e_1, \ldots)$ over alternating *States*(A) and *Events*(A) such that $s_0 \in$ *Initial*(A) and (s_i, s_{i+1}) is a transition of e_i for all i. A finite sequence σ over alternating *States*(A) and *Events*(A) may end in a state or an event. A finite behavior, on the other hand, ends in a state by definition. The set of behaviors of A is denoted by *Behaviors*(A). The set of finite behaviors of A is denoted by *Finite Behaviors*(A).

For $e \in$ *Events*(A), let *enabled*$_A(e)$ be the set $\{s$: for some state t, $(s, t) \in$ *Transitions*$_A(e)\}$. An event e is said to be enabled in a state s of A iff $s \in enabled_A(e)$. An event e is said to be disabled in a state s of A iff $s \notin enabled_A(e)$.

Notation. For any sequence σ over alternating $States(A)$ and $Events(A)$, and for any set $E \subseteq Events(A)$, $image(\sigma, E)$ denotes the sequence of events over E obtained from σ by deleting states and deleting events that are not in E.

Definition. A module M is defined by:

- $Events(M)$, a set of events that is the union of three disjoint sets:
  $Inputs(M)$, a set of input events,
  $Outputs(M)$, a set of output events, and
  $Internals(M)$, a set of internal events.
- $sts(M)$, a state transition system with Events$(sts(M)) = Events(M)$.
- *Fairness requirements of* M, a finite collection of subsets of $Outputs(M) \cup Internals(M)$. Each subset is referred to as a fairness requirement of M.

Convention. For readability, the notation $sts(M)$ is abbreviated to M wherever such abbreviation causes no ambiguity, e.g., $States(sts(M))$ is abbreviated to $States(M)$, $enabled_{sts(M)}(e)$ is abbreviated to $enabled_M(e)$, etc.

Let F be a fairness requirement of module M. F is said to be enabled in a state s of M iff, for some $e \in F$, e is enabled in s. In a behavior $\sigma = (s_0, e_0, s_1, e_1, \ldots, s_j, e_j, \ldots)$, we say that F occurs in state s_j iff $e_j \in F$. An infinite behavior σ of M satisfies F iff F occurs infinitely often or is disabled infinitely often in states of σ.

For module M, a behavior σ is an *allowed behavior* iff for every fairness requirement F of M: σ is finite and F is not enabled in its last state, or σ is infinite and satisfies F. Let $AllowedBehaviors(M)$ denote the set of allowed behaviors of M.

We are now in a position to formalize the notion of a *module offers an interface*. Consider module M and interface I. Let σ be a sequence over alternating states and events of module M.

Definition. σ is allowed wrt I iff image $(\sigma, Events(I)) \in AllowedEventSeqs(I)$.

Definition. σ is safe wrt I iff one of the following holds:

- σ is finite and $image(\sigma, Events(I)) \in SafeEventSeqs(I)$.
- σ is infinite and every finite prefix of σ is safe wrt I.

In what follows, we use $last(\sigma)$ to denote the last state in finite behavior σ, and @ to denote concatenation.

Definition. Given a module M and an interface I, M *offers* I iff the following conditions hold:

- Naming constraints:
  $Inputs(M) = Inputs(I)$ and $Outputs(M) = Outputs(I)$.
- Safety constraints:
  For all $\sigma \in FiniteBehaviors(M)$, if σ is safe wrt I, then
  $\forall e \in Outputs(M)$:
  $last(\sigma) \in enabled_M(e) \Rightarrow \sigma @ e$ is safe wrt I, and

  $\forall e \in Inputs(M)$:
  $\sigma @ e$ is safe wrt $I \Rightarrow last(\sigma) \in enabled_M(e)$.
- Progress constraints:
  For all $\sigma \in AllowedBehaviors(M)$,
  if σ is safe wrt I, then σ is allowed wrt I.

Note that module M is required to satisfy interface I only if its environment satisfies the safety requirements of I. Specifically, for any finite behavior that is not safe wrt I, the two Safety constraints are satisfied trivially; for any allowed behavior of M that is not safe wrt I, the Progress constraint is satisfied trivially. That is, as soon as the environment of M violates some safety requirement of I, module M can behave arbitrarily and still satisfy the definition of M *offers* I.

The two Safety constraints can be stated informally as follows: First, whenever an output event of M is enabled to occur, the event's occurrence would be safe, i.e., if the event occurs next, the resulting sequence of interface event, occurrences is a prefix of an allowed event sequence of I. Second, whenever an input event of M (controlled by its environment) can safely, M does not block the event's occurrence.

For an input event of M whose occurrence would be unsafe, module M has a choice: it may block the event's occurrence or let it occur. (In this respect, our model is more general than the I/O automata model [15, 16], which requires an I/O automation to be always input-enabled.)

2.2 Module composition

A module M with upper interface U and lower interface L is illustrated in Fig. 1. The environment of M consists of the user of U and the module that offers L. In what follows, we use "σ is safe wrt U and L" to mean "σ is safe wrt U and σ is safe wrt L", and ϕ to denote the empty set.

Definition. Given module M and interfaces U and L, M *using* L *offers* U iff the following conditions hold:

- Naming constraints:

  $Events(U) \cap Events(L) = \emptyset$,
  $Inputs(M) = Inputs(U) \cup Outputs(L)$, and

  $Outputs(M) = Outputs(U) \cup Inputs(L)$.
- Safety constraints:
  For all $\sigma \in FiniteBehaviors(M)$,
  if σ is safe wrt U and L, then
  $\forall e \in Outputs(M)$:
  $last(\sigma) \in enabled_M(e)$
  $\Rightarrow \sigma @ e$ is safe wrt U and L, and

  $\forall e \in Inputs(M)$:
  $\sigma @ e$ is safe wrt U and L
  $\Rightarrow last(\sigma) \in enabled_M(e)$.
- Progress constraints:
  For all $\sigma \in AllowedBehaviors(M)$,
  if σ is safe wrt U and L, then
  σ is allowed wrt $L \Rightarrow \sigma$ is allowed wrt U.

The definition of M *using* L *offers* U is similar to the definition of M *offers* I in most respects. The main difference between the two definitions is in the Progress constraints. For module M using interface L, it is required to satisfy the progress requirements of interface U *only if* the module that offers L satisfies the progress requirements of L.

We next define how modules are composed. Our definition is like the one by Lynch and Tuttle [15], with the exception that we hide output events that match input events.

Definition. A finite set of modules $\{M_j : j \in J\}$ are compatible iff $\forall j, k \in J, j \neq k$:

$Internals(M_j) \cap Events(M_k) = \emptyset$, and
$Outputs(M_j) \cap Outputs(M_k) = \emptyset$.

Notation. For a set of modules $\{M_j : j \in J\}$, each state of their composition is a tuple $s = (t_j : j \in J)$, where $t_j \in States(M_j)$. We use $image(s, M_j)$ to denote t_j.

Definition. Given a compatible set of modules $\{M_j : j \in J\}$, their composition is a module M defined as follows:

- $Events(M)$ defined by:
$$Internals(M) = [\bigcup_{j \in J} Internals(M_j)] \cup [(\bigcup_{j \in J} Outputs(M_j)) \cap (\bigcup_{j \in J} Inputs(M_j))]$$
$$Outputs(M) = [\bigcup_{j \in J} Outputs(M_j)] - [\bigcup_{j \in J} Inputs(M_j)]$$
$$Inputs(M) = [\bigcup_{j \in J} Inputs(M_j)] - [\bigcup_{j \in J} Outputs(M_j)]$$

- $sts(M)$ defined by:
$$States(M) = \prod_{j \in J} States(M_j)$$
$$Initial(M) = \prod_{j \in J} Initial(M_j)$$
$Transitions_M(e)$, for all $e \in Events(M)$, defined by: $(s, t) \in Transitions_M(e)$ iff, $\forall j \in J$,
if $e \in Events(M_j)$ then $(image(s, M_j), image(t, M_j)) \in Transitions_{M_j}(e)$, and
if $e \notin Events(M_j)$ then image $(s, M_j) = image(t, M_j)$.

- *Fairness requirements of* M $= [\bigcup_{j \in J}$ *Fairness requirements of* $M_j]$.

Theorem 1 *Let modules, M and N, and interfaces, U and L, satisfy the following*:
- $Internals(M) \cap Internals(N) = \emptyset$
- *M using L offers U*
- *N offers L*

Then, M and N are compatible and their composition offers U.

A proof of Theorem 1 can be found in [10]. It is quite long, requiring the proof of several lemmas.

Theorem 2 *Let $M_1, I_1, M_2, I_2, \ldots, M_n, I_n$ be a finite sequence over alternating modules and interfaces, such that the following hold*:
- *For all j, k, if $j \neq k$ then $Events(I_j) \cap Events(I_k) = \emptyset$ and $Internals(M_j) \cap Events(M_k) = \emptyset$.*
- *M_1 offers I_1.*
- *For $j = 2, \ldots, n$, M_j using I_{j-1} offers I_j.*

Then, modules $M_1, \ldots, M_n$ are compatible and their composition offers I_n.

Proof. The compatibility of $\{M_1, \ldots, M_n\}$ is obvious. To show that the composition offers I_n, it suffices to establish the following inductive step, for $j = 2, \ldots, n$:

> If the composition of $\{M_1, \ldots, M_{j-1}\}$ offers I_{j-1}, and M_j using I_{j-1} offers I_j, then the composition of $\{M_1, \ldots, M_{j-1}, M_j\}$ offers I_j.

But this is implied by Theorem 1, with the composition of $\{M_1, \ldots, M_{j-1}\}$ being N, M_j being M, I_{j-1} being L, and I_j being U. □

2.3 Relational notation

In this section, we introduce the relational notation for specifying state transition systems, modules and interfaces. The notation has two basic constructs: state formulas that represent sets of states, and event formulas that represent sets of state transitions [9]. The definitions and results of Sects. 2.1 and 2.2 are recast in this notation.

The state space of a state transition system is specified by a set of variables, called state variables. For a state transition system A, the set of variables is denoted by $Variables(A)$. For each variable v, there is a set $domain(v)$ of allowed values. By definition, $States(A) = \prod_{v \in Variables(A)} domain(v)$. Each state $s \in States(A)$ is represented by a tuple of values, $(d_v : v \in Variables(A))$, where $d_v \in domain(v)$.

We use state formulas to represent subsets of $States(A)$. A *state formula* is a formula in $Variables(A)$ that evaluates to true or false when $Variables(A)$ is assigned s, for every state $s \in States(A)$. A state formula represents the set of states for which it evaluates to true. For state s and state formula P, s satisfies P iff P evaluates to true for s.[1]

We use event formulas to specify the transitions of events. An *event formula* is a formula in $Variables(A) \cup Variables(A)'$, where $Variables(A)' = \{v' : v \in Variables(A)\}$ and $domain(v') = domain(v)$. The ordered pair $(s, t) \in States(A) \times States(A)$ is a transition specified by an event formula iff (s, t) satisfies the event formula, that is, the event formula evaluates to true when $Variables(A)$ is assigned s and $Variables(A)'$ is assigned t.

Definition. A state transition system A is specified in the relational notation by:

- $Events(A)$, a set of events.
- $Variables(A)$, a set of state variables, and their domains.

[1] We use *formula* to mean a *well-formed formula* in the language of predicate logic

- $Initial_A$, a state formula specifying the initial states.
- For every event $e \in Events(A)$, an event formula $formula_A(e)$ specifying the transitions of e.

Note that for each event e, we have

$$enabled_A(e) = [\exists Variables(A)' : formula_A(e)]$$

which is a state formula representing the set of states where e is enabled.

Definition. A module M is specified in the relational notation by:

- Disjoint sets of events, $Inputs(M)$, $Outputs(M)$, and $Internals(M)$, with $Events(M)$ being their union.
- $sts(M)$, a state transition system with $Events(sts(M)) = Events(M)$, specified in the relational notation.
- *Fairness requirements of* M, a finite collection of subsets of $Outputs(M) \cup Internals(M)$.

To specify an interface in the relational notation, we use a state transition system together with invariant and progress assertions. In what follows, we first introduce the assertions and then explain how the allowed event sequences of an interface are specified.

Invariant assertions are of the form: *invariant P*, where P is a state formula. A finite sequence over alternating states and events satisfies *invariant P* iff every state in the sequence satisfies P. An infinite sequence over alternating states and events satisfies *invariant P* iff every finite prefix of the sequence satisfies *invariant P*.

We use leads-to assertions of the form: *P leads-to Q*, where P and Q are state formulas.[2] A sequence $(s_0, e_0, s_1, e_1, \ldots)$ over alternating states and events satisfies *P leads-to Q* iff for all i: if s_i satisfies P then there exists $j, j \geqq i$, such that s_j satisfies Q.

Invariant and leads-to assertions are collectively referred to as atomic assertions. In what follows, an *assertion* is either an atomic assertion or one constructed from atomic assertions using logical connectives and quantifiers.

Let σ denote a sequence over alternating states and events. An atomic assertion is true for σ iff σ satisfies the assertion. The truth value of a nonatomic assertion, say *Assert*, is determinated by first evaluating for σ the truth value of every atomic assertion within *Assert*. For example, σ satisfies the assertion $X \wedge Y \Rightarrow Z$, where X, Y and Z are atomic assertions, iff $(\sigma$ satisfies $X) \wedge (\sigma$ satisfies $Y) \Rightarrow (\sigma$ satisfies $Z)$.

A safety assertion is an assertion constructed from invariant assertions only. A state transition system satisfies a safety assertion iff every finite behavior of the state transition system satisfies the safety assertion. A *progress assertion* is an assertion constructed from atomic assertions that include at least one leads-to assertion. A module satisfies a progress assertion iff every allowed behavior of the module satisfies the progress assertion.

For brevity, we often write assertions and rules containing free occurrences of parameters. We follow the convention that such assertions and rules are universally quantified over all values of the free parameters. For example, the assertion, $x = k$ *leads-to* $x = k+1$, has x as a state variable and k as a parameter. This assertion is equivalent to $[\forall k : x = k$ *leads-to* $x = k+1]$.

To use a state transition system, say A, for specifying an interface, we need to exercise care in defining the events of A. To see why, an interface, called I_A, defined as follows:

- $Inputs(I_A) = Inputs(A)$, $Outputs(I_A) = Outputs(A)$ and
- $AllowedEventSeqs(I_A)$ $= \{image(\sigma, Inputs(A) \cup Outputs(A)) : \sigma \in Behaviors(A)\}$.

Think of A as a module with no fairness requirement. In general A does not offer I_A. It is easy to see that A offers I_A if every transition of A is identified by a distinct event. Such a condition, however, is a very strong requirement. We provide a weaker condition that is sufficient for *A offers* I_A.

Definition. A state transition system A has *deterministic events* iff

- $Internals(A) = \emptyset$,
- $Initial(A)$ is a single state, and
- *for all* $e \in Events(A)$, $Transitions_A(e)$ is a partial function, i.e., for all $s \in States(A)$, there is at most one state s' such that $(s, s') \in Transitions_A(e)$.

This condition is easy to satisfy because events in our theory can be regarded as names or labels. (Moreover, event names can be parameterized in the relational notation [9].) Each event sequence represents at most one behavior of A because event occurrences have deterministic effects. Behaviors of A, however, are nondeterministic because more than one event can be enabled in a state. (In part II of [10], the above condition is relaxed to allow the use of internal events.)

Note that the restriction of a single initial state can be circumvented as follows (if needed): Let s_0 denote a state not in $States(A)$, and $Init(A)$ the desired initial states of A. Definite $Initial(A)$ to be $\{s_0\}$ and, for all $s \in Init(A)$, specify a distinct event for each transition (s_0, s).

Notation. For any state formula R, we use R' to denote the formula obtained from R by replacing every state variable v in it with v'.

Definition. An interface I is specified in the relational notation by:

- Disjoint sets of events, $Inputs(I)$ and $Outputs(I)$, with $Events(I)$ being their union.
- $sts(I)$, a state transition system with deterministic events specified in the relational notation such that $Events(sts(I)) = Events(I)$.[3]

[2] *Leads-to* is the only temporal connective we use

[3] For readability, the notation $sts(I)$ is abbreviated to I wherever such abbreviation causes no ambiguity, e.g., $Variables(sts(I))$ is abbreviated to $Variables(I)$, $formula_{sts(I)}(e)$ is abbreviated to $formula_I(e)$, etc.

- $InvAssum_I$, a conjunction of state formulas referred to as *invariant assumptions* of I, such that
  $Initial_I \Rightarrow InvAssum_I$, and
  $\forall e \in Outputs(I)$: $InvAssum_I \wedge formula_I(e) \Rightarrow InvAssum'_I$
- $InvGuar_I$, a conjunction of state formulas referred to as *invariant guarantees* of I, such that
  $Initial_I \Rightarrow InvGuar_I$, and
  $\forall e \in Inputs(I)$: $InvGuar_I \wedge formula_I(e) \Rightarrow InvGuar'_I$
- $ProgReqs_I$, a conjunction of progress assertions, referred to as *progress requirements* of I.

The invariant assumptions and guarantees of interface I are collectively referred to as *invariant requirements* of interface I. Define[4]

$InvReqs_I \equiv InvAssum_I \wedge InvGuar_I$.

Given an interface I specified in the relational notation, an allowed event sequence of I is the sequence of events in a behavior of $sts(I)$ that satisfies all invariant and progress requirements; more precisely, define

$AllowedBehaviors(I)$
$= \{\sigma: \sigma \in Behaviors(I)$ and σ satisfies *invariant* $InvReqs_I$ and $ProgReqs_I\}$, and
$AllowedEventSeqs(I)$
$= \{image(\sigma, Events(I)): \sigma \in AllowedBehaviors(I)\}$.

Lastly, for event $e \in Events(I)$, define

$possible_I(e) \equiv InvReqs_I$
$\wedge [\exists Variables(I)': formula_I(e) \wedge InvReqs'_I]$

which is a state formula representing the set of states in which event e can occur without violating any safety requirement of I.

Note that 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 allowed state transitions encoded in a state transition system. Our approach is a flexible one, including the following as special cases: (1) Safety requirements of I are specified using a state transition system only, namely $sts(I)$, without any invariant requirement. Satisfaction of the safety requirements of I by a module M is proved by showing that $sts(M)$ is a refinement of $sts(I)$; definition of refinement is given below. (2) The state transition system $sts(I)$ has a single state variable, namely, a "trace" variable that records the sequence of all event occurrences. Each event of $sts(I)$ is always enabled and each event's action is to update the trace variable. In this case, safety requirements are specified exclusively by invariant requirements that are predicates on event traces.

[4] In the latest version of our method, presented in part II of [10], it is no longer required that the invariant requirements of an interface be partitioned into assumptions and guarantees. Furthermore, the **B** and **C** conditions in Sections 2.4 have been modified and relaxed

2.4 Module composition in relational notation

For modules and interfaces specified in the relational notation, we provide sufficient conditions for *M offers I* and *M using L offers U*. We first introduce a refinement relation between two state transition systems A and B such that $Variables(A) \supseteq Variables(B)$. In this case, there is a projection mapping from $States(A)$ to $States(B)$ defined as follows: state $s \in States(A)$ is mapped to state $t \in States(B)$ where t is defined by the values of $Variables(B)$ in s [7, 9, 23]. State formulas in $Variables(B)$ can be interpreted directly over $States(A)$ using the projection mapping. Also, event formulas in $Variables(B) \cup Variables(B)'$ can be interpreted directly over $States(A) \times States(A)$ using the projection mapping.

Definition. Given state transition systems A and B and state formula Inv_A in $Variables(A)$, A is a refinement of B assuming Inv_A iff

- $Variables(A) \supseteq Variables(B)$ and $Events(A) \supseteq Events(B)$
- $Initial_A \Rightarrow Initial_B$
- $\forall e \in Events(B)$: $Inv_A \wedge formula_A(e) \Rightarrow formula_B(e)$ (event refinement condition)
- $\forall e \in Events(A) - Events(B)$:
  $Inv_A \wedge formula_A(e) \quad \Rightarrow [\forall v \in Variables(B): v = v']$ (null image condition)

If A is a refinement of B assuming Inv_A and, moreover, A satisfies *invariant* Inv_A, then A is a refinement of B as defined in [9]. In this case, for any state formula P in $Variables(B)$, if B satisfies *invariant* P, then A satisfies *invariant* P.

Given a module M, an interface I, and some state formula Inv_M in $Variables(M)$, the following conditions are sufficient for *M offers I*:

B1 $Inputs(M) = Inputs(I)$ and $Outputs(M) = Outputs(I)$
B2 $sts(M)$ is a refinement of $sts(I)$ assuming Inv_M
B3 $\forall e \in Inputs(I)$: $Inv_M \wedge possible_I(e) \Rightarrow enabled_M(e)$
B4 $\forall e \in Outputs(I)$: $Inv_M \wedge formula_M(e) \Rightarrow InvGuar'_I$
B5 $sts(M)$ *satisfies* (*invariant* $InvAssum_I \Rightarrow$ *invariant* Inv_M)
B6 M satisfies (*invariant* $InvAssum_I \Rightarrow ProgReqs_I$)

Condition **B1** is the same as the Naming constraints in *M offers I*. **B1**, **B2**, **B4** and **B5** imply the following,

$\forall \sigma \in Behaviors(M)$: σ satisfies *invariant* $InvAssum_I$ iff σ is safe wrt I.

B2, **B4** and **B5** ensure that M satisfies the safety requirements of I assuming *invariant* $InvAssum_I$ (first Safety constraint in *M offers I*). **B3** and **B5** ensure that M does not block the occurrence of any input event whenever the event can occur safely (second Safety constraint in *M offers I*). Progress constraints in *M offers I* hold because **B6** ensures that if an allowed behavior of M satisfies *invariant* $InvAssum_I$, it satisfies $ProgReqs_I$.

Theorem 3 *For a module M, an interface I, and some state formula Inv_M in Variables(M), if conditions* **B1–B6** *hold, then M offers I.*

Given an interface I, to obtain a module M that offers I, we make use of **B1–B6** in three stages. First, the events of $sts(M)$ are named such that **B1** is satisfied. Second, events of $sts(M)$ are specified such that $sts(M)$ is a refinement of $sts(I)$ (**B2** is satisfied), each input event is enabled in states where the event's occurrence would be safe (**B3** is satisfied), and M satisfies its invariant guarantees (**B4** is satisfied). Initially, Inv_M is set equal to $InvAssum_I$. But to prove **B2–B4**, we may have to assume that $sts(M)$ has additional invariant properties, which are used to strengthen Inv_M and must be proved (so that **B5** is satisfied). Lastly, we try to prove **B5** and **B6**. (Some useful inference rules are given in Sect. 2.5.)

For a module M, interfaces U and L, and some state formula Inv_M in *Variables*(M), the following conditions are sufficient for *M using L offers U*:

C1 $Events(U) \cap Events(L) = \emptyset$
$Inputs(M) = Inputs(U) \cup Outputs(L)$
$Outputs(M) = Outputs(U) \cup Inputs(L)$
$Variables(U) \cap Variables(L) = \emptyset$

C2 $sts(M)$ is a refinement of $sts(U)$ assuming Inv_M

C3 $sts(M)$ is a refinement of $sts(L)$ assuming Inv_M

C4 $\forall e \in Inputs(U)$: $Inv_M \wedge possible_U(e) \Rightarrow enabled_M(e)$

C5 $\forall e \in Outputs(L)$: $Inv_M \wedge possible_L(e) \Rightarrow enabled_M(e)$

C6 $\forall e \in Inputs(L)$: $Inv_M \wedge formula_M(e) \Rightarrow InvAssum'_L$

C7 $\forall e \in Outputs(U)$: $Inv_M \wedge formula_M(e) \Rightarrow InvGuar'_U$

C8 $sts(M)$ satisfies $(invariant(InvAssum_U \wedge InvGuar_L) \Rightarrow invariant\ Inv_M)$

C9 M satisfies $(invariant(InvAssum_U \wedge InvGuar_L) \wedge ProgReqs_L \Rightarrow ProgReqs_U)$

Theorem 4. *For a module M, interfaces U and L, and some state formula Inv_M in Variables(M), if conditions* **C1–C9** *hold, then M using L offers U.*

C8 indicates that we can set Inv_M equal to $InvAssum_U \wedge InvGuar_L$ initially. However, to prove **C2–C7** for a module M, we may have to assume that $sts(M)$ has additional invariant properties, which are used to strengthen Inv_M and must be proved (so that **C8** is satisfied).

Conditions **C2, C3** and **C8** specify that module M must block every input event occurrence that would violate any safety requirement encoded in $sts(U)$ or $sts(L)$. (While such blocking is allowed by the semantic definition of *M using L offers U*, it is not required.) As a result, **C1–C3** and **C6–C8** imply the following,

$\forall \sigma \in Behaviors(M)$:

σ satisfies $invariant(InvAssum_U \wedge InvGuar_L)$ iff σ is safe wrt U and L.

In this respect, conditions **C1–C9** are stronger than the semantic definition of *M using L offers U*; similarly, conditions **B1–B6** are stronger than the semantic definition of *M offers I*. The **B** and **C** conditions are applicable to the database examples in this paper. However, for applications in general, it is desirable to relax them as much as possible (see part II of [10] for relaxed conditions).

2.5 Conventions, auxiliary variables and inference rules

We review in this section some features of the relational notation to be used in the database examples of Sects. 3–6. (See [9] for a more thorough treatment.)

Conventions for event formulas

An event formula defines a set of state transitions. Some examples of event definitions are shown below:

$e_1 \equiv v_1 > 2 \wedge v'_2 \in \{1, 2, 5\}$
$e_2 \equiv v_1 > v_2 \wedge v_1 + v'_2 = 5$

In each definition, the event name is given on the left-hand side of "$\equiv$" and the event formula is given on the right-hand side.

Consider a state transition system A with two state variables v_1 and v_2. Let e_2 above be an event of the system. Note that v'_1 does not occur free in *formula*(e_2). By the following convention, it is assumed that v_1 is not updated by the occurrence of e_2.

Convention. Given an event formula, *formula*(e), for every state variable v in *Variables*(A), if v' is not a free variable of *formula*(e), the conjunct $v' = v$ is implicit in *formula*(e).

If a parameter occurs free in an event's formula, then there is an event defined for every allowed value of the parameter. For example, consider

$e_3(m) \equiv v_1 > v_2 \wedge v_1 + v'_2 = m$

where m is a parameter with a specified domain of allowed values. A parameterized event is a convenient way to specify a set of related events.

Lastly, in deriving a state transition system A from a state transition system B, for A to be a refinement of B as defined in Sect. 2.4, we further require that every parameter of B be a parameter of A with the same name and same domain of allowed values.

Auxiliary variables

For a module M, some of its state variables in *Variables*(M) may be *auxiliary variables*-i.e., state variables that are needed for specification or verification only, and do not have to be included in an implementation of the module.[5] Informally, a subset of variables in *Variables*(M) is auxiliary if they do not affect the enabling condition of any event nor do they affect the update of any state

[5] What we call auxiliary variables here are also known as history variables. Abadi and Lamport showed that another kind of auxiliary variables, called prophecy variables, is needed for a refinement method, such as ours, to be *complete* [1]

variable that is not auxiliary [20]. To state the above condition precisely, let $Auxvars(M)$ be a proper subset of $Variables(M)$, and $Auxvars(M)' = \{v' : v \in Auxvars(M)\}$. The state variables in $Auxvars(M)$ are auxiliary if, for every event e of $sts(M)$, the following holds:

$$formula_M(e) \Rightarrow [\forall Auxvars(M) \exists Auxvars(M)' : formula_M(e)]$$

If the above condition is satisfied, $Auxvars(M)$ do not have to be implemented. More precisely, let N be a module that is an implementation of M, defined as follows:

- $Variables(N) = Variables(M) - Auxvars(M)$, with the same domain for each variable as in M
- $Initial_N = [\exists Auxvars(M): Initial_M]$
- $Events(N) = Events(M)$, with the same partition into input, output and internal events
- *Fairness requirements of N* $=$ *Fairness requirements of M*
- for every event $e \in Events(N)$: $formula_N(e) \equiv [\forall Auxvars(M) \exists Auxvars(M)' : formula_M(e)]$

It is shown in [9] that N is a well-formed image of M such that the following hold:[6]

$$\{image(\sigma, N) : \sigma \in Behaviors(M)\} = Behaviors(N)$$
$$\{image(\sigma, N) : \sigma \in AllowedBehaviors(M)\} = AllowedBehaviors(N)$$

where $image(\sigma, N)$ denotes the observation of a behavior σ of M when auxiliary variables are invisible. That is, $image(\sigma, N)$ denotes a sequence over alternating $States(N)$ and $Events(N)$ obtained from σ as follows: each state s in σ is replaced by its image s' using the projection mapping from $States(M)$ to $States(N)$. Thus, modules M and N cannot be distinguished by observations when auxiliary variables in M are invisible. In particular, the following result is presented in [10]: For any two interfaces L and U, M using L offers U if and only if N using L offers U.

State functions

In the database examples below, we will also use *state functions*-namely, 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.

Inference rules

To facilitate proofs of invariant and leads-to assertions in the database examples below, we present some inference rules.

Invariance rule: State transition system A satisfies *invariant P* if

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

Note that if A satisfies *invariant* I then $I \wedge I'$ can be used to strengthen the antecedent of the logical implication above, i.e., replace P by $I \wedge I' \wedge P$. Also if A satisfies *invariant* P and $P \Rightarrow Q$, for state formula Q, then A satisfies *invariant* Q.

Definition. For module M that includes F as a fairness requirement, *P leads-to Q via F* iff

(i) for every event e in F, $P \wedge formula_M(e) \Rightarrow Q'$,
(ii) for every event f of M, $P \wedge formula_M(f) \Rightarrow P' \vee Q'$, and
(iii) $invariant[\exists e \in F : P \Rightarrow enabled(e)]$.

Some inference rules for leads-to assertions are given below.[7]

Leads-to rules: *P leads-to Q* if one of the following holds:

- *invariant* $P \Rightarrow Q$ [implication]
- for some fairness requirement F, *P leads-to Q via F* [event]
- for some state formula 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]
- *invariant I* and $P \wedge I$ *leads-to Q* [substitution]

3 Serializable database interface U

The problem posed by Lamport [13] is to specify a serializable database interface, and also specify an implementation of a database system that satisfies the interface specification. There is a set of client programs that use the database system. It is assumed that client programs execute concurrently; each issues a sequence of transactions to be processed by the database system.

In this paper, the serializable database interface is called interface U, which is specified in this section. Specification of a lower interface L for accessing a physical database is given in Sect. 4. An implementation of a database system is specified as a module. We present two modules below. Module M_{TPL}, based upon the method of two-phase locking, is specified in Sect. 5; we prove that M_{TPL} using L offers U. Module M_{MVT}, based upon the method of multi-version timestamps, is specified in Sect. 6; we prove that M_{MVT} offers U.

Lamport's informal specification of an interface consists of a set of procedures that can be executed concurrently by transactions of different client programs [13]. We model such an interface procedure P by two events: $Call(P)$ and $Return(P)$. Since several invocations of P

[6] The key result is Lemma 12 in [9]

[7] For a comprehensive treatment of proof rules, the reader is referred to [4, 17, 21]. For distributed systems with unreliable communication channels, see [9] for the *P leads-to Q via Msg* rule, where *Msg* denotes a set of messages

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 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 modules, each procedure is also modeled two events $Call(i, P)$ and $Return(i, P)$, which are obtained by refining the *matching* interface events – that is, interface events of the same names. Because the action of each event in our formalism is atomic, the atomic actions of modules may be too large in the following sense: Consider a module implemented using 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 of the module are updated by events in the procedure body. The call and return events are used to transfer control and parameter values only. Thus for implementation in a practical programming language, each module presented 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 into auxiliary variables. In Sect. 7, we indicate how such refinements can be carried out.

Interface U is specified in Sects. 3.1–3.5 below. Before doing so, we define several constants. Let OBJECTS denote the set of objects in a database, VALUES the set of values each object can have, KEYS a finite 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.[8] 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 use NULL to denote a special value that is not in any of the sets, KEYS, VALUES, OBJECTS, and IDS.

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

3.1 State variables of interface U

H: sequence of {(*id*, *Begin*, *key*), (*id*, *Read*, *key*, *obj*, *val*), (*id*, *Write*, *key*, *obj*, *val*, OK), (*id*, *End*, *key*, 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} ∪ {(*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 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 be 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*) ∈ H means that H has a (*id*, *Read*, *obj*, *key*, *val*) or a (*id*, *Write*, *obj*, *key*, *val*, OK) entry for some *key* and some *val*.

We next introduce notation that is used in our definition of serializability below. 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*) ∉ h.
= *val*, if (*obj*) ∈ h and (*obj*, *val*) is the last such entry.

3.2 State functions of interface U

We first define two state functions that are used to specify when H is serializable.

comids: powerset of IDS
The set of committed transactions.
Formally, *comids* = {*id* : (*id*, *End*) ∈ H}.

[8] If a transaction can have multiple procedure calls outstanding, a tuple (*id*, *n*) would be needed – instead of *id* alone – to identify an interface procedure call

H_serializable: boolean

True iff there is a permutation $id_1, id_2, \ldots, id_{|comids|}$ of the elements in *comids* such that

$S = H(id_1) @ H(id_2) @ \ldots @ H(id_{|comids|})$ satisfies

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

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 database systems. The two-phase locking module satisfies conflict serializability, the strongest condition of the three. However, the multi-version timestamp module satisfies only multi-version view serializability, the weakest condition of the three. The above definition, a form of multi-version view serializability, can be used for both modules to be specified in this paper.

We next define state functions that are used to specify when transactions are in conflict and when keys are used incorrectly.

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*.

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

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), $(id, Abort)$, $(i, Abort) \notin h$.

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

$=$ NULL, if $\neg active(id)$.

$= key$, if $active(id)$ and $(id, Begin, key)$ is the first $(id, Begin)$ entry in H.

correctkeyuse: boolean.

True iff every transaction has used the correct key in all its procedure calls. Formally,

$correctkeyuse =$ true iff

$((id, key) \in H \vee status(id) = (key)) \Rightarrow key = keyof(id)$.

3.3 Events of interface U

For readability, we model each procedure return by one of two possible return events, one for success and one for abort.

Call$(id, Begin)$

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

Return$(id, Begin, key)$

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

Return$(id, Begin, \text{FAILED})$

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

Call$(id, Read, key, obj)$

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

Return$(id, Read, key, obj, val)$

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

Return$(id, Read, key, obj, \text{ABORT})$

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

Call$(id, Write, key, obj, val)$

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

Return$(id, Write, key, obj, val, \text{OK})$

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

Return$(id, Write, key, obj, val, \text{ABORT})$

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

Call(id, End, key)

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

Return$(id, End, key, \text{OK})$

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

Return$(id, End, key, \text{ABORT})$

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

Call$(id, Abort, key)$

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

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

3.4 Safety requirements of interface U

Many safety requirements stated informally by Lamport [13] are specified implicitly in $sts(U)$. Consider the event formulas in Sect. 3.3. First, the informal requirement that each client program must wait for the return from a procedure call before issuing another call is specified by including the conjunct $status(id) = \text{READY}$ in the enabling condition of a call event, and updating $status(id)$ to a value not equal to READY in the action of the call event. Only the action of a return event can change the value of $status(id)$ back to READY. Furthermore, one $status(id)$ has been updated to the value ABORTED or COMMITED, no more calls can be issued by the transaction with identifier *id*.

An invocation of *Begin* is always enabled to return FAILED. This is weaker than Lamport's requirement that an invocation of *Begin* should return FAILED only when there are insufficient resources to start another transaction [13], e.g., when there is no unallocated key. However, Lamport's requirement cannot be modeled because he does not provide any information on what resources are needed to start a transaction (other than keys). If such information is known, then a state formula defining the condition of "insufficient resources" can be included in the enabling condition of the return event.

The informal requirement that invocations of *Read*, *Write*, *End* and *Abort* can be made only with a key of an active transaction is specified by the conjunct $allocated(key)$ in the enabling condition of each such call event. The reuse of keys is specified by the conjunct $\neg allocated(key)'$ in an *Abort* or *End* return event.

Lastly, 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 module, this condition can be strengthened by requiring both *id* and *i* to be currently active.)

There are two safety requirements of interface *U* that are specified as invariant requirements, one is an assumption and the other a guarantee:

$InvAssum_U \equiv correctkeyuse$
$InvGuar_U \equiv H\_serializable$

By definition, we have

$InvReqs_U \equiv correctkeyuse \wedge H\_serializable.$

3.5 Progress requirements of interface U

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

$$R_1 \equiv status(id) \in \{(Begin), (Read), (Write), (End), (Abort)\} \; \textit{leads-to} \; status(id) \in \{\text{READY, ABORTED, COMMITTED, NOTBEGUN}\}$$

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:

$$R_2 \equiv (status(id) \in \{(Read), (Write)\} \; \textit{leads-to} \; status(id) = \text{READY}) \Rightarrow (status(id) = \text{READY} \; \textit{leads-to} \; status(id) = (End))$$

The following progress requirement is specified for interface U:

$ProgReqs_U \equiv [\forall id : R_2] \Rightarrow R_1.$

4 Physical database interface L

The two-phase locking module M_{TPL}, to be specified in Sect. 5, uses a lower interface *L* for accessing a physical database. Interface *L* is specified below. Note that outstanding procedure calls at the lower interface are uniquely identified by the entries of KEYS.

4.1 State variables of interface L

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

4.2 State functions of interface L

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

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

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

deadlock(*key*, *obj*): boolean.

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

4.3 Events of interface L

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

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

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

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

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

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

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

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

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

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

4.4 Safety requirements of interface L

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. Interface *L* has no invariant requirement.

$InvReqs_L \equiv \text{true}$

4.5 Progress requirements of interface L

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

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

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

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

$$Q_4 \equiv R_4 \Rightarrow G_4$$

where

$$R_4 \equiv [\forall k_2 : waiting(k_1, obj) \wedge owned(k_2, obj) \textit{ leads-to } waiting(k_1, obj) \wedge \neg owned(k_2, obj)]$$

$$G_4 \equiv waiting(k_1, obj) \textit{ leads-to } owned(k_1, obj)$$

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 [13].

$$Q_5 \equiv cycle(k_1, k_2, \ldots, k_n) \textit{ leads-to } [\exists i, 1 \leq i \leq n : status_L(k_i) = \text{READY}]$$

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

$ProgReqs_L \equiv Q_1 \wedge Q_2 \wedge Q_3 \wedge Q_4 \wedge Q_5.$

5 Two-phase locking module M_{TPL}

The two-phase locking module M_{TPL} makes use of interface *L* to offer interface *U*. The state transition system of M_{TPL} is obtained from interfaces *U* and *L* by adding new state variables, and refining the events of *U* and *L*. Note that we choose to have a module that does not block any incorrect use of allocated keys.

5.1 State variables of M_{TPL}

In addition to the state variables *H*, *status*, and *allocated* of interface *U*, and the state variables $status_L$, *owned*, and *storedvalue* of interface *L*, 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 $\{(id, Begin, key), (id, Read, key, obj, val), (id, Write, key, obj, val, \mathrm{OK}), (\mathrm{id, End, key, OK})\}$. 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 of interface U 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.

5.2 *State functions of* M_{TPL}

holdinglocks(key): boolean.

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

5.3 *Events and refinement requirements of* M_{TPL}

Module events that match the events of interface U are listed first. (These module events have null images at the lower interface because they do not update any state variable of the lower interface.) The formulas of these module events are obtained by refining formulas of the matching events of interface U. For most events, the formula of a module event e is obtained by adding conjuncts to the formula of interface event e. Below, we use ⟨interface formula⟩ to denote the formula of the matching interface event given in Sect. 3.3. When the refinement is not of this simple form, we add a condition which must be implied by Inv_M.

$Call(id, Begin) \equiv \langle\text{interface formula}\rangle$

$Return(id, Begin, key) \equiv \langle\text{interface formula}\rangle \wedge \neg holdinglocks(key)$

$Return(id, Begin, \mathrm{FAILED}) \equiv \langle\text{interface formula}\rangle$

$Call(id, Read, key, obj) \equiv \langle\text{interface formula}\rangle$

$Return(id, Read, key, obj, val) \equiv$
$\quad \langle\text{interface formula}\rangle$
$\quad \wedge\, localvalue(obj, key) \neq \mathrm{NULL}$
$\quad \wedge\, val = localvalue(obj, key)$

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

For the above event and the matching interface event to satisfy the event refinement condition assuming Inv_M, it is sufficient that Inv_M implies the following:

$status(id) = (obj) \wedge aborting(key) \Rightarrow concurrentaccess(id)$

The above requirement is satisfied by assuming the following condition and *correctkeyuse* (to be conjuncts of Inv_M):

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

$Call(id, Write, key, obj, val) \equiv \langle\text{interface formula}\rangle$

$Return(id, Write, key, obj, val, \mathrm{OK})$
$\quad \equiv \langle\text{interface formula}\rangle \wedge locked(key, obj)$
$\quad \wedge\, localvalue(obj, key)' = val$

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

Assuming $A_1 \wedge correctkeyuse$, the above event and the matching interface event satisfy the event refinement condition.

$Call(id, End, key) \equiv \langle\text{interface formula}\rangle$

$Return(id, End, key, \mathrm{OK})$
$\quad \equiv \langle\text{interface formula}\rangle$
$\quad \wedge\, [\forall x\colon localvalue(x, key) = \mathrm{NULL}]$
$\quad \wedge\, S' = S\,@\,H(id)$

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

$Call(id, Abort, key) \equiv \langle\text{interface formula}\rangle$

$Return(id, Abort, key)$
$\quad \equiv \langle\text{interface formula}\rangle$
$\quad \wedge\, [\forall x\colon localvalue(x, key)' = \mathrm{NULL}]$

We next define module events that match events of the lower interface L. For all of these module events, the formula of each is obtained by adding conjuncts to the formula of the matching lower interface event.

For every lower interface event, say f, defined in Sect. 4.3, the event is *renamed* to be the same as the matching module event. For convenience, we use f to denote the formula of the interface event in defining its matching module event. Below, each module event is defined by a formula of the form $formula(e) = f \wedge p$, where f denotes the formula of the matching lower interface event and p is some event formula in state variables of the module that are not state variables of the upper or lower interfaces. This special form ensures that module event e is a refinement of the matching lower interface event, and it has a null image at the upper interface.

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

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

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

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

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

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

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

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

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

Note that a module event is classified as an input (output) event iff it matches a call (return) event of the upper interface or a return (call) event of the lower interface. This completes our specification of the events of module M_{TPL}. Note also that the module has no internal events.

5.4 *Fairness requirements of* M_{TPL}

We specify the following fairness requirements for module M_{TPL} (events *RequestLock*, *RequestRead*, *RequestWrite*, and *ReqRelLock* are called *request* events):

F1: For each return event e of the module, there is a fairness requirement consisting of e.

F2: For each request event e of the module, there is a fairness requirement consisting of e.

This completes our specification of module M_{TPL}.

5.5 *Informal description of two-phase locking*

We provide below an informal description of the two-phase locking module, by indicating the sequence of event occurrences for each transaction call. Those who are familiar with the two-phase locking protocol might want to skip ahead to Sect. 5.6. 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 module 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 module assigns *val* to *localvalue*(*obj*, *key*) and executes *Return*(*Write*, 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 module 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 module sets *localvalue*(*obj*, *id*) to *val* and executes *Return*(*Write*, 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 module 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 module executes *RequestLock*(*key*, *obj*), which is eventually followed by either *LockAcquired*(*key*, *obj*) or *LockRejected*(*key*, *obj*). In the first case, the module 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 module 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 module executes a *Return*(*End*, OK), ending the transaction's execution. The second stage, referred to as *lock-releasing*, then follows. In this stage, the module returns all of the locks acquired by the transaction. For each *obj* such that *locked*(*key*, *obj*) is true, the module 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 module returns the locks acquired by the transaction, exactly as in the lock-releasing stage following a *Call*(*End*, *key*).

5.6 *Proof that two-phase locking module using L offers U*

We apply Theorem 4 to prove that M_{TPL} using L offers U. It is sufficient to establish that conditions **C1–C9** in Sect. 2.4 are satisfied.

Satisfaction of conditions C1–C6

Suppose the lower interface events have been renamed to be the same as their matching module events. From the fact that the upper and lower interfaces have no state variable in common, condition **C1** is satisfied.

The state variable set of the module includes all state variables of the upper interface, with the same initial conditions. Each module event that matches an upper interface event has been constructed so that it is a refinement of the interface event-assuming $A_1 \wedge correctkeyuse$, in two cases – and has a null image on the lower interface. Thus, condition **C2** is satisfied assuming $A_1 \wedge correctkeyuse$. (Note that Inv_M in condition **C8** must imply $A_1 \wedge correctkeyuse$.)

The state variable set of the module includes all state variables of the lower interface, with the same initial conditions. Each module event that matches a lower interface event has been constructed so that it is a refinement of the interface event, and has a null image on the upper interface. Thus, condition **C3** is satisfied.

Condition **C4** is satisfied because, for every call event of interface U, the formula of the matching module event is identical to the formula of the call event.

Condition **C5** is satisfied because, for every return event of the lower interface, the formula of the matching module event has the form $f \wedge p$, where f is the formula of the interface return event and p is such that $enabled(p)$ is true.

Condition **C6** is satisfied vacuously because the lower interface has no invariant requirements, i.e., $InvAssum_L = \text{true}$.

Satisfaction of conditions C7–C8

For satisfaction of condition **C7**, we need to show that every output event of M_{TPL} preserves $InvGuar_U$, which is *H_serializable*. Recall that S denotes a serial history obtained by concatenating histories of committed transactions in the order of commitment, that is, $S = H(id_1) @ H(id_2) @ \ldots @ H(id_{|comids|})$, where $id_1, id_2, \ldots, id_{|comids|}$ denote identifiers of the committed transactions in the order of commitment. From the definition of *H_serializable* in Sect. 3.2, it suffices to show that every output event of M_{TPL} preserves the following:

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

The $Return(id, End, \text{OK})$ event is the only event that can affect A_2. It concatenates $H(id)$ to the end of S. Thus A_2 is preserved by every output event of M_{TPL} if the following condition holds just before each occurrence of the $Return(id, End, \text{OK})$ event.

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

Thus, **C7** is satisfied if Inv_M implies A_3. In addition, to ensure that events of module M_{TPL} are refinements of events of interface U (from condition **C2**), Inv_M must imply $correctkeyuse \wedge A_1$. Thus, if condition **C8** is proved for $Inv_M \equiv correctkeyuse \wedge A_1 \wedge A_3$, both conditions **C2** and **C7** are satisfied.

To show that **C8** is satisfied, we present a proof that M_{TPL} satisfies (*invariant* $correctkeyuse \Rightarrow$ *invariant* $A_1 \wedge A_3$). We present here an informal justification of the invariance of A_1 and A_3. A more formal proof is given in Appendix A.

We first consider A_1, which is

$$\begin{aligned} A_1 \equiv\ & keyof(id) = key \wedge status(id) = (obj) \wedge aborting(key) \\ & \Rightarrow concurrentaccess(id). \end{aligned}$$

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*(*id*). 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.

We next consider A_3. To establish its invariance, 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. These values are related during the growing and committing stages of a transaction by the following conditions, which we assert to be invariant:

$$\begin{aligned} A_4 \equiv\ & (\forall key: \neg locked(key, obj)) \\ & \Rightarrow storedvalue(obj) = lastvalue(obj, S) \end{aligned}$$

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

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

$$\begin{aligned} A_7 \equiv\ & keyof(id) = key \wedge locked(key, obj) \\ & \wedge status(id) = (End) \\ & \Rightarrow (id, obj) \in H \\ & \wedge \text{(a) } [(localvalue(obj, key) = lastvalue(obj, H(id)) \\ & \quad \wedge storedvalue(obj) = lastvalue(obj, S)) \\ & \text{(b) } \vee (localvalue(obj, key) = \text{NULL} \\ & \quad \wedge storedvalue(obj) = lastvalue(obj, H(id)))] \end{aligned}$$

An informal justification of the above invariant assertions follows (formal proof in Appendix A). 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).

In what follows, we use the notation A_{i-j} to denote $A_i \wedge A_{i+1} \wedge \ldots \wedge A_j$.

Lemma 1. *M_{TPL} satisfies*
(invariant correctkeyuse $\Rightarrow$ invariant $A_1 \wedge A_{4-7}$).
Proof in Appendix A.

It remains for us to prove that A_3 is 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.

Satisfaction of condition C9

Recall that module M_{TPL} has fairness requirements **F1** and **F2** (defined in Sect. 5.4). Also, we can assume that the module satisfies progress requirements Q_1, Q_2, Q_3, Q_4, Q_5 of the lower interface. We proceed to prove that the module satisfies the progress requirement of interface U. Throughout *invariant correckeyuse* is assumed to be satisfied.

Lemma 2. *Module M_{TPL} satisfies the following progress assertions:*

$W_1 \equiv status(id) = (Begin)$
  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*, 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-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-to $status(id) = (End, key)$
  $\wedge\ (\forall obj\colon localvalue(obj, key) = \text{NULL})$

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

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

From *Return*(*Write*, OK), we have:
$status(id) = (Write, key, obj) \wedge locked(key, obj)$
  leads-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). □

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* $\neg waiting(k_1, obj)$

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 a directed graph defined by nodes KEYS $\cup$ OBJECTS and edges $\{(x, k): owned(k, x)\} \cup \{(k, x): waiting(k, x)\}$. 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 $k_1, x_1, k_2, x_2, \ldots, k_J$ be the sequence of *distinct* nodes such that $waiting(k_i, x_i)$ and $owned(k_{i+1}, x_i)$ for $1 \leqq i < J$, and k_J satisfies one of the following three mutually exclusive conditions:

$unblocked(k_J)$ $\equiv k_J$ is not waiting for any object.
$blocked(k_J)$ $\equiv k_J$ is waiting for an object that is not locked by any key.
$deadlocked(k_J) \equiv k_J$ is waiting for an object locked by k_i for some i, $1 \leqq i < J$.

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 indicate the last key on the path. At any time, $1 \leqq J \leqq |\text{KEYS}|$. Every k_i, $1 \leqq i < J$, 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[9], where $1 \leqq i \leqq |\text{KEYS}|$. We use M to denote a symbol that is greater than any integer; that is, for any integer n, $n < M$ holds.

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 $\neg allocated(k_J)$.
$= 0$, 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 $\Delta = (\beta_1, \alpha_2, \beta_2, \alpha_3, \ldots, \alpha_{|\text{KEYS}|}, \beta_{|\text{KEYS}|})$. The values of Δ are totally-ordered lexicographically. We shall prove the following:

$W_8 \equiv waiting(k_1, x_1) \wedge \Delta = \mathbf{a}$
leads-to $\neg waiting(k_1, x_1) \vee \Delta > \mathbf{a}$

We first show that W_7 follows from W_8. W_8 states 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. (Note that β_i increasing without bound implies that R_4, the antecedant of Q_4, is true.) The latter is not allowed by R_2, the assumption that every transaction accesses at most a finite number of objects. Thus, it suffices to prove W_8.

Lemma 3. *Module M_{TPL} satisfies the following progress assertion*:

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

Proof. Assume $J = j$ and *allocated* (k_J), that is, k_j is releasing its locks. $J = j$ and $\Delta = \mathbf{a}$ hold until k_j releases its lock on x_{j-1}. At this point, W_{9b} holds with $J = j - 1$ and $\Delta > \mathbf{a}$. 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 R_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 $\Delta = \mathbf{a}$. If the object is locked by some key already on the path, (that is, by k_i for some i, $1 \leqq i < j$), then W_{9c} results with $J = j$ and $\Delta = \mathbf{a}$. If the object is locked by some key not already on the path, then the path gets extended, resulting in $\Delta > \mathbf{a}$; specifically, J becomes $l > j$, and α_i increases from 0 for $j < i \leqq l$. 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. □

Lemma 4. *Module M_{TPL} satisfies the following progress assertion*:

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

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 $\Delta > \mathbf{a}$ 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 $\Delta > \mathbf{a}$ because J becomes $j + 1$ and α_{j+1} increases from 0. □

[9] For reasoning using proof rules, these functions can be replaced by appropiately-defined auxiliary variables

Lemma 5. *Module M_{TPL} satisfies the following progress assertion*:

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

Proof. Assume $J=j$. Then we have a cycle consisting of k_j and other (perhaps all) keys in the path. $LockRejected(k_i, x_j)$ 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 $1 \leqq l \leqq j$. If $l=1$, we get $unblocked(k_1)$. If $l>1$, then J becomes l, α_i and β_i become 0 for $l<i\leqq j$, and α_l increases to M. $\Delta > \mathbf{a}$ holds because α_l is lexicographically more significant than α_i or β_i, for $l<i\leqq j$. □

From the implication rule, we have

$blocked(k_J) \wedge \Delta > \mathbf{a}\ leads\text{-}to\ \Delta > \mathbf{a}$.

Using the disjunction rule on the above and W_{10}, we get

$W_{12} \equiv blocked(k_J) \wedge \Delta \geqq \mathbf{a}\ leads\text{-}to\ unblocked(k_1) \vee \Delta > \mathbf{a}$

Similarly, from W_{11}, we can infer

$W_{13} \equiv deadlocked(k_J) \wedge \Delta \geqq \mathbf{a}$
$\quad leads\text{-}to\ unblocked(k_1) \vee \Delta > \mathbf{a}$

W_{12} has the form W_{9b} *leads-to* Z, and W_{13} has the form W_{9c} *leads-to* Z, where $Z \equiv unblocked(k_1) \vee \Delta > \mathbf{a}$. W_9 has the form X *leads-to* $W_{9a} \vee W_{9b} \vee W_{9c}$, where $X \equiv unblocked(k_J) \wedge \Delta = \mathbf{a}$. Applying the transitivity and disjunction rules to W_9, W_{12}, and W_{13} (with W_{12} at W_{9b}, and W_{13} at W_{9c}), we get X *leads-to* $W_{9a} \vee Z \vee Z$, which can be simplified to

$unblocked(k_J) \wedge \Delta = \mathbf{a}\ leads\text{-}to\ unblocked(k_1) \vee \Delta > \mathbf{a}$.

Applying the disjunction rule to this, W_{10}, and W_{11}, we get W_8, noting that $unblocked(k_1) \Rightarrow \neg waiting(k_1, x_1)$ and $unblocked(k_J) \vee blocked(k_J) \vee deadlocked(k_J) \equiv$ true. Recall that W_8 is sufficient for W_7, and W_1–W_7 are sufficient for module M_{TPL} to satisfy the progress requirement of interface U.

6 Multi-version timestamp module M_{MVT}

A module, M_{MVT}, that implements the multi-version timestamp protocol is specified in this section. It offers the serializable database interface U (specified in Sect. 3). Unlike the two-phase locking module, module M_{MVT} does not use a lower interface. But like the two-phase locking module, we choose to specify M_{MVT} such that it does not block any incorrect use of allocated keys. Before specifying M_{MVT}, we provide an informal overview of the multi-version timestamp protocol below.

Module M_{MVT} uses "timestamps" that are nonnegative integers. For notational consistency with the specification of interface U, timestamps will be referred to as keys. For each object, the module 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 $[ov.wkey \dots ov.rkey]$ constitute the *interval* of *ov*, where $[i \dots j]$ denotes the set $\{i, i+1, \dots, j\}$. While a version *ov* of an object exists, no transaction whose key is in $[ov.wkey \dots ov.rkey-1]$ 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: $[ov_1.wkey \dots ov_1.rkey-1] \cap [ov_2.wkey \dots ov_2.rkey-1] = \{\,\}$. 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.

6.1 *State variables of M_{MVT}*

In addition to the state variables *H*, *status*, and *allocated* of interface U, 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 $\{ov: ov.value = \mathrm{INITVALUE}(obj),$
$ov.wkey = ov.rkey = 0\}$.
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 $\{(id, Begin, key), (id, Read, key, obj, val), (id, Write, key, obj, val, \mathrm{OK}), (id, End, key, \mathrm{OK})\}$.
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 interface U 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 $S(>key)_i$.

6.2 Events and refinement requirements of M_{MVT}

The following definition is used in the module events below:

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

The module events are listed below. Each module event matches an event of interface U. Formulas of the module events are obtained by refining the formulas of matching interface U events. Below, we use ⟨interface formula⟩ to denote the formula of the matching interface event given in Sect. 3.3.

$Call(id, Begin) \equiv$ ⟨interface formula⟩

$$\begin{aligned} Return(id, Begin, key) \equiv\ & \langle\text{interface formula}\rangle \\ & \wedge key = laststarted' = laststarted + 1 \\ & \wedge dependsupon(key)' = \{\ \} \end{aligned}$$

$Return(id, Begin, \text{FAILED}) \equiv$ ⟨interface formula⟩

$Call(id, Read, key, obj) \equiv$ ⟨interface formula⟩

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

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.

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

For the above event and the matching interface U event to satisfy the event refinement condition assuming Inv_M, it is sufficient that Inv_M implies the following:

$$\begin{aligned} & status(id) = (obj) \wedge dependsupon(key) \cap aborted \neq \{\ \} \\ & \qquad \Rightarrow concurrentaccess(id) \end{aligned}$$

The above requirement is satisfied by assuming the following condition and $correctkeyuse$ (to be conjuncts of Inv_M):

$$\begin{aligned} B_1 \equiv\ & keyof(id) = key \wedge dependsupon(key) \cap aborted \neq \{\ \} \\ & \Rightarrow concurrentaccess(id) \end{aligned}$$

$Call(id, Write, key, obj, val) \equiv$ ⟨interface formula⟩

$$\begin{aligned} & Return(id, Write, key, obj, val, \text{OK}) \\ & \quad \equiv \langle\text{interface formula}\rangle \\ & \quad \wedge dependsupon(key) \cap aborted = \{\ \} \\ & \quad \wedge \neg [\exists ov \in versions(obj): \\ & \qquad key \in [ov.wkey \ldots ov.rkey - 1]] \\ & \quad \wedge versions(obj)' = \{ov_1 \in versions(obj): \\ & \qquad ov_1.wkey \neq key\} \\ & \qquad \cup \{ov_2: ov_2.value = val \\ & \qquad \wedge ov_2.wkey = ov_2.rkey = key\} \end{aligned}$$

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

The above event and the matching interface U event satisfy the event refinement condition assuming $B_1 \wedge B_2 \wedge correctkeyuse$, where

$$\begin{aligned} B_2 \equiv\ & keyof(id) = key \wedge status(id) = (obj) \\ & \wedge ov \in versions(obj) \\ & \wedge key \in [ov.wkey \ldots ov.rkey - 1] \\ & \quad \Rightarrow concurrentaccess(id) \end{aligned}$$

$Call(id, End, key) \equiv$ ⟨interface formula⟩

$$\begin{aligned} & Return(id, End, key, \text{OK}) \equiv \langle\text{interface formula}\rangle \\ & \quad \wedge dependsupon(key) \subseteq done - aborted \\ & \quad \wedge done' = done \cup \{key\} \\ & \quad \wedge S' = S(<key) @ H(key) @ S(>key) \end{aligned}$$

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

The above event and the matching interface U event satisfy the event refinement condition assuming $B_1 \wedge correctkeyuse$.

$Call(id, Abort, key) \equiv$ ⟨interface formula⟩

$$\begin{aligned} Return(id, Abort, key) \equiv\ & \langle\text{interface formula}\rangle \\ & \wedge Abort(key) \end{aligned}$$

Note that a module event is an input (output) event iff it matches a call (return) event of the interface. This completes our specification of the module events. Note that module M_{MVT} has no internal events.

6.3 *Fairness requirements of* M_{MVT}

We assume the following fairness requirements for module M_{MVT}:

F1: For each return event e of the module, there is a fairness requirement consisting of e.

This completes our specification of module M_{MVT}.

6.4 *Proof that multi-version timestamp module offers U*

We apply Theorem 3 to prove that M_{MVT} offers U. It is sufficient is to establish that conditions **B1–B5** are satisfied.

Satisfaction of conditions B1–B3

It is obvious that condition **B1** is satisfied.

The state variable set of module M_{MVT} includes all state variables of interface U with the same initial conditions. Each module event has been constructed so that it is a refinement of the matching interface event–assuming *correctkeyuse* in conjunction with B_1 and B_2 in some cases. Thus, condition **B2** is satisfied for some Inv_M that implies $B_1 \wedge B_2 \wedge correctkeyuse$.

Condition **B3** is satisfied because, for every call event of interface U, the formula of the matching module event is identical to the formula of the call event.

Satisfaction of conditions B4–B5

For satisfaction of condition **B4**, we need to show that every output event of M_{MVT} preserves $InvGuar_U$, which is *H_serializable.* Recall that $S = H(id_1)\ @\ H(id_2)\ @\ \dots\ @\ H(id_{|\text{comids}|})$, where id_1, $id_2, \dots, id_{|\text{comids}|}$ denote the identifiers of committed transactions in the order of their timestamps. From the definition of *H_serializable* in Sect. 3.2, it suffices to show that every output event of M_{MVT} preserves the following:

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

Note that the *Return*(*End*, *key*, OK) event is the only event that can affect B_3. Specifically, it inserts $H(key)$ between $S(<key)$ and $S(>key)$. Thus B_3 is preserved by every output event of M_{MVT} if the following conditions hold just before each occurrence of the *Return*(*End*, *key*, OK).

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

$$\begin{aligned} B_5 \equiv\ & key \notin done \wedge dependsupon(key) \subseteq done - aborted \\ & \wedge H(key)_i = (Read, obj, val) \wedge (obj) \in H(key)_{<i} \\ & \Rightarrow val = lasvalue(obj, H(key)_{<i}) \end{aligned}$$

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

Thus, **B4** is satisfied if Inv_M implies B_{4-6}. In addition, to ensure that events of module M_{MVT} are refinements of events of interface U (condition **B2** above), Inv_M must imply $correctkeyuse \wedge B_{1-2}$. Thus, if condition **B5** is proved for $Inv_M = correctkeyuse \wedge B_{1-2} \wedge B_{4-6}$, both conditions **B2** and **B4** are satisfied.

To show that **B5** is satisfied, we sketch a proof that M_{MVT} satisfies (*invariant correctkeyuse*$\Rightarrow$*invariant* $B_{1-2} \wedge B_{4-6}$). We first provide an informal justification of the invariance of B_1 and B_2, repeated here. (Additional invariant requirements needed for a formal proof are indicated below).

$$\begin{aligned} B_1 \equiv\ & keyof(id) = key \wedge dependsupon(key) \cap aborted \neq \{\ \} \\ & \Rightarrow concurrentaccess(id) \end{aligned}$$

$$\begin{aligned} B_2 \equiv\ & keyof(id) = key \wedge status(id) = (obj) \\ & \wedge ov \in versions(obj) \\ & \wedge key \in [ov.wkey \dots ov.rkey - 1] \\ & \Rightarrow concurrent\text{-}access(id) \end{aligned}$$

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*, *x*), which read from a version ov of some object x with $ov.wkey = k_1$. Clearly, transaction id was active and accessing object x when this event occurred. Transaction id_1 had accessed object x and was either active or committed when the event occurred, because version 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 object x. 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*) = (*obj*). 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).

Let us now examine B_4, B_5 and B_6. 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 $H(key)$. These various values are related during execution of the transaction using *key* by the following conditions, which we assert to be invariant:

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

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

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

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

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

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

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 values equals to 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 $ov.value = val$, $key \in [ov.wkey+1 \ldots ov.rkey]$, 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 $k \in [ov.wkey \ldots ov.rkey-1]$, 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 $k \in [ov.wkey+1 \ldots ov.rkey]$ 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, $key \in [ov.wkey+1 \ldots ov.rkey-1]$ 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 transition 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. *M_{MVT} satisfies (invariant correctkeyuse$\Rightarrow$ invariant$B_{1-2} \wedge B_{7-12}$).*

Proof omitted.

To prove the above lemma formally, by showing that the assertion satisfies the invariance rule, additional invariant assertions are needed.

Recall that invariance of B_{7-12} is sufficient for invariance of B_{4-6}. Thus Lemma 6 is sufficient for satisfaction of **B5**.

Satisfaction of condition B6

Recall that module M_{MVT} has fairness requirements **F1**. We next prove that module M_{MVT} satisfies the progress requirement of interface U assuming that *correctkeyuse* is invariant.

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

Lemma 7. *Module M_{MVT} satisfies the following progress assertions:*

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

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

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

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

$$X_5 \equiv lastdone = j \wedge laststarted > j \text{ leads-to } lastdone > j$$

Proof. X_1 and X_2 are proved exactly as W_1 and W_3 are proved for the two-phase locking module (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 $lastdone \geqq dependsupon(key)$. 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 R_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. □

Applying the transitivity rule repeatedly to X_5, we get $status(id) = (End, key)$ *leads-to* $lastdone \geqq dependsupon(key)$. Combining this with X_4, we get $status(id) = (End)$ *leads-to* $status(id) \in \{\text{COMMITTED, ABORTED}\}$. Applying the disjunction rule to this and X_1 through X_3, module M_{MVT} satisfies the progress requirement of interface U.

7 Implementation of procedure calls

Lamport's informal specification of a module interface consists of a set of procedures [13]. 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 module events, however, *nonauxiliary* state variables can be updated as part of the atomic action of $Return(id, P)$. For example, given an interface procedure P with input parameters $\mathbf{x}$ and result parameters $\mathbf{y}$, we have module 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 $\mathbf{v}$ 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 module specifications in this paper need to be refined further. We briefly discuss how such a refinement can be carried out without actually doing it, under the assumption that procedure bodies do not interfere with each other [20].

We can make all variables in $\mathbf{v}$ into auxiliary variables, and introduce additional state variables $\mathbf{u}$. Let $\mathbf{w}$ be a subset of $\mathbf{u}$ 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 $\mathbf{u}$. 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 $\mathbf{u}$, we need to introduce a set of events $\{body_i,\ i = 1, \ldots, n\}$ that constitute the body of the procedure P. Each such event has the following form:

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

where b_i is a boolean function of $\mathbf{u}$ and $\mathbf{u}' = h_i(\mathbf{u})$ represents a computation that the new module can perform as an atomic action. Observe that each $body_i$ event satisfies the null image condition for the new module to be a refinement of the old module. These events perform updates specified by function g in the old event, but in n atomic actions instead of one.

The above approach is similar to one suggested by Lamport [11], where he transforms the nonauxiliary state variables in $\mathbf{v}$ of the old module into auxiliary state functions of the new module.

8 Concluding remarks

An interface between a module and its environment is defined by a set of allowed sequences of interface events. This is like specifications of CSP processes [5] and I/O automata [15, 16]. However, other than this, our theory and the theories of CSP and I/O automata are quite different.

In the theory of CSP, the semantics of a process is defined by a set of finite traces and associated refusal sets; in our theory, the semantics of a module is defined by a set of behaviors and a set of fairness requirements (each behavior is represented by a sequence of alternating states and events). Specifically, the concepts of internal state and fairness are essential in our theory but are absent in the theory of CSP. Also, there is no requirement in the CSP model that interface events are partitioned into inputs and outputs. Such a requirement is essential for formalizing our notion of a two-sided interface between a service provider and service consumer, i.e., defining what it means to offer an interface and to use an interface (see [10] for a more in-depth comparison).

In the theory of I/O automata [15, 16], there is no distinction between module and interface, service provider and service consumer. There is the notion of one automaton simulating another automaton, but not our notion of a two-sided interface. Furthermore, each I/O automaton is required to be input-enabled, i.e., every input event is enabled in every state of the automaton. In this respect, our model is more general; a module in our theory is required to be input-enabled *only when* the occurrence of an input event would not violate any safety requirement of the module's interface(s). For an input event whose occurrence would be unsafe, the module has a choice: it may disable the input or let it occur.

The model of Abadi and Lamport [2] is state-based, without interface events. In this respect, it is fundamentally different from our model and those in [5, 15].

A restriction in our model that is uniquely our is that modules can only be composed hierarchically. We accepted this restriction because we were motivated by our interest in decomposing the specification of a complex system (such as the protocols of a computer network) rather than the kind of composition problems of interest in the area of distributed algorithms.

To specify nontrivial examples, we prefer to use the relational notation [9]. We find it more convenient to work with state formulas and event formulas than individual states and transitions, and to reason with invariant and progress assertions than safe and allowed even sequences.

In relational specifications, the set of allowed sequences of interface events is not represented directly. Instead, a labeled state transition system and a set of invariant and progress requirements are specified, and the set of allowed sequences is obtained from event sequences in the allowed behaviors of the state transition system. Having states represented explicitly in behaviors facilitates our proof that a module offers an interface. Specifically, we make use of a projection mapping from module states to interface states to prove that the state transition systems of the module and interface satisfy a refinement relation. By using auxiliary variables, our projection mappings [9] are as general as multi-valued possibilities mappings [15].

Conceptually, the use of a state transition system in an interface specification should not influence an implementor, because only the set of allowed event sequences, generated by the state transition system and constrained by the assertions, is of interest. In practice, however, the state transition system might bias implementors of modules that offer the interface.

Acknowledgements. We thank the anonymous referees for their constructive criticisms. One referee was so thorough and insightful that this paper is much improved not only in presentation style but also in substance. With the permission of the Managing Editor, we gratefully acknowledge Michael Merritt of Bell Laboratories for his thorough reviews and constructive insights. Additionally, the writing of this paper has benefited from numerous discussions with Sandra Murphy of the University of Maryland, College Park. We also thank Leslie Lamport of DEC, and Ken Calvert, Mohamed Gouda and Thomas Woo of the University of Texas at Austin for their helpful comments. In particular, Ken Calvert suggested the condition of deterministic events for relationally-specified interfaces.

References

1. Abadi M, Lamport L: The existence of refinement mappings. Research Report 29, Digital Systems Research Center, Palo Alto, CA 94301, August 1988
2. Abadi M, Lamport L: Composing specifications. In: de Bakker, JW, de Roever W-P, Rozenberg G (eds) Stepwise refinement of distributed systems. LNCS vol 430. Springer, Berlin Heidelberg New York 1990
3. Bernstein PA, Hadzilacos V, Goodman N: Concurrency control and recovery in database systems. Addison-Wesley, Reading, Massachusetts 1987
4. Chandy KM, Misra J: A foundation of parallel program design. Addison-Wesley, Reading, Massachusetts 1988
5. Hoare CAR: Communicating sequential processes. Prentice-Hall, Englewood Cliffs, NJ 1985
6. Jonsson B: On decomposing and refining specifications of distributed systems. In: de Bakker JW, de Roever W-P, Rozenberg G (eds) Stepwise refinement of distributed systems. LNCS vol 430. Springer, Berlin Heidelberg New York 1990
7. Lam SS, Shankar AU: Protocol verification via projections. IEEE Trans. Software Eng. Vol. SE-10, 10: 325–342 (1984)
8. Lam SS, Shankar AU: 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
9. Lam SS, Shankar AU: A relational notation for state transition systems. IEEE Trans. Software Eng 16(7):755–775 (1990) (an abbreviated version entitled Refinement and Projection of Relational Specifications. In: de Bakker W, de Roever W-P, Rozenberg G (eds) Stepwise refinement of distributed systems) LNCS vol 430. Springer, Berlin Heidelberg New York 1990
10. Lam SS, Shankar AU: A theory of interfaces and modules, part I and part II. Technical reports, Department of Computer Sciences, University of Texas at Austin, 1992. An abbreviated version of part I entitled *Understanding Interfaces*. In: Proceedings IFIP Fourth International Conference on Formal Description Techniques (FORTE), Sydney, Australia, November 1991
11. Lamport L: An assertional correctness proof of a distributed algorithm. Sci Comput Program 2: 175–206 (1982)
12. Lamport L: What it means for a concurrent program to satisfy a specification: why no one has specified priority. Proceedings 12th ACM Symposium on Principles of Programming Languages. New Orleans 1985
13. Lamport L: A serializable database interface. 26th Lake Arrowhead Workshop on how will we specify concurrent systems in the year 2000. September 1987
14. Lamport L: A simple approach to specifying concurrent systems. Comm ACM 32 (1):32–45 (1989)
15. Lynch N, Tuttle M: Hierarchical correctness proofs for distributed algorithms. Proceedings of the ACM Symposium on Principles of Distributed Computing, Vancouver, B.C., August 1987
16. Lynch N, Merritt M, Weihl W, Fekete A: A theory of atomic transactions. Technical Report MIT/LCS/TM-362, Laboratory for Computer Science, M.I.T, June 1988

17. Manna Z, Pnueli A: Adequate proof principles for invariance and liveness properties of concurrent programs. Sci Comput Program 4:257–289 (1984)
18. Misra J, Chandy KM: Proofs of networks of processes. IEEE Trans Software Eng. Vol. SE-7, 4: 417–426 (1981)
19. Murphy SL, Shankar AU: Service specification and protocol construction for the transport layer, CS-TR-2033, UMIACS-TR-88-38, Computer Science Department University of Maryland, May 1988; an abbreviated version appears in Proc. ACM SIGCOMM '88 Symposium, August 1988
20. Owicki S, Gries D: An axiomatic proof technique for parallel programs I. Acta Inf 6: 319–340 (1976)
21. Owicki S, Lamport L: Proving liveness properties of concurrent systems. ACM TOPLAS 4(3):455–495 (1982)
22. Pnueli A: In transition from global to modular temporal reasoning about programs. NATO ASI Series. In: Logics and models of concurrent systems. Apt KR (ed) vol F13, Springer, Berlin Heidelberg New York 1984. pp 123–144
23. Shankar AU, Lam SS: An HDLC protocol specification and its verification using image protocols. ACM Trans Comput Syst. Vol. 1, 4: 331–368 (1983)
24. Shankar AU, Lam SS: A stepwise refinement heuristic for protocol construction. To appear in ACM TOPLAS; an abbreviated version entitled Construction of Network Protocols by Stepwise Refinement. In: de Bakker JW, de Roever W, Rozenberg G (eds): Stepwise refinement of distributed systems. LNCS vol 430. Springer, Berlin Heidelberg New York 1990

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:

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

$A_9 \equiv allocated(key) \Rightarrow [\exists\, exactly\ one\ id\colon keyof(id) = key]$

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.

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

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

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

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

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

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

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

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

The following formulas specify relationships when a transaction is aborting:

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

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

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

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

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

$A_7 \equiv keyof(id) = key \wedge locked(key, obj) \wedge status(id) = (End)$
$\quad \Rightarrow (id, obj) \in H$
$\quad\quad \wedge\ [(a)\ (localvalue(obj, key) = lastvalue(obj, H(id))$
$\quad\quad\quad\quad \wedge\, storedvalue(obj) = lastvalue(obj, S))$
$\quad\quad\quad \vee (b)\ (localvalue(obj, key) = \text{NULL}$
$\quad\quad\quad\quad \wedge\, storedvalue(obj) = lastvalue(obj, H(id)))]$

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

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

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

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

The following are also needed:

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

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

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

Lemma A2. *$A_1 \wedge A_{4-7} \wedge A_{10-23}$ satisfies the invariance rule, given that $A_8 \wedge A_9$ is invariant.*

Proof omitted.