ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 51 (2004) 47–85

Science of Computer Programming

www.elsevier.com/locate/scico

Using eternity variables to specify and prove a serializable database interface

Wim H. Hesselink*

Department of Mathematics and Computing Science, Rijksuniversiteit Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands

Received 9 January 2003; received in revised form 6 May 2003; accepted 5 June 2003

Abstract

Eternity variables are introduced to specify and verify serializability of transactions of a distributed database. Eternity variables are a new kind of auxiliary variables. They do not occur in the implementation but are used in specification and verification. Elsewhere it has been proved that eternity variables in combination with history variables are semantically complete for proving refinement relations.

An eternity variable can be thought of as an unknown constant that is determined by the behaviour (execution sequence). In the specification of the database, one eternity variable is used to enforce serialization. In the verification, an additional eternity variable is needed for the connection of the local data with the shared database.

The formalism is based on linear-time temporal logic, but the analysis of behaviours is completely reduced to the next-state relation together with progress arguments using variant functions. Forward invariants (inductive predicates) are complemented with other, so-called backward, invariants. The proof has been verified with the first-order theorem prover NQTHM to give additional confidence in the result and in the feasibility of the approach.

Keywords: Serializability; Specification; Implementation; History variables; Prophecy variables; Forward invariant; Backward invariant; Mechanical verification

1. Introduction

The 26th Lake Arrowhead Workshop, held in September 1987, was devoted to the question: "How will we specify concurrent systems in the year 2000?" The participants

* Tel.: +31-50-3633933; fax: +31-50-3633800.
E-mail address: wim@cs.rug.nl (W.H. Hesselink).
URL: http://www.cs.rug.nl/∼wim

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2003.06.001

were provided with an informal description of a serializable database interface and invited to present a formal specification at the meeting. The workshop resulted in five papers published in Distributed Computing 6, 1992. Schneider [18] introduces the setting and the informal description. There are three proposed solutions by Broy [4], Kurki-Suonio [10], and Lam and Shankar [11]. Finally, Lamport [12] discusses the solutions. He argues that verification methods should allow for the prophecy variables introduced by Abadi and Lamport in [1], since that is the way to get a semantically complete method. Yet, the combination of history variables and prophecy variables is proved to be complete in [1] only under certain finiteness assumptions. Prophecy variables are not very well known or often used.

Our approach to the database problem indeed asks for variables with some kind of prescient capabilities. The prophecies needed are a choice of a value for the database and of a transaction number. Since prophecy variables with infinite choices are unsound [1], application of prophecy variables would require to specify beforehand that the state space of the database is finite. This is a heavy condition: since the state space is constant, finiteness of the state space implies boundedness. It would therefore require not only that the database always has finite contents, but even that the number of objects that can be stored is bounded. It would also require bounds on the transaction numbers, while we would prefer to allow the database to support infinitely many transactions.

We rejected the option to change the specification in order to ease the proof of correctness of the implementation. Instead of this, we invented another kind of auxiliary variables with "prescient" behaviour, so-called eternity variables. In [7,8], we show that the verification method based on eternity variables in combination with forward simulations is semantically complete, in a slightly stronger sense than the combination of prophecy variables and history variables of [1].

There is nothing magical about eternity variables. An eternity variable is just an auxiliary variable that is initialized nondeterministically and that is never modified. Eternity variables have two kinds of roles. In specifications, the value of the eternity variable together with a supplementary property can be used to rule out unwanted behaviours. In verifications of refinement, its value is constrained by a relation with the state, the so-called behaviour restriction, which can be used as an invariant provided that it is satisfiable for every behaviour of the program.

The behaviour restriction is responsible for the aspects that may seem prophetic to the operationally reasoning programmer. The soundness of extension with eternity variables is a rather easy consequence of the behaviour restriction. The complications of the prophecy variables of [1] are mainly due to the choice to rely on König's lemma rather than on something like a behaviour restriction. Indeed, the behaviour restriction may look like a heavy and unmanageable condition, but in this paper we show that it can be used effectively to handle the classical practical problem of the serializable database interface.

Overview: In Section 2, we develop the formal computational model. We generalize (weaken) the concept of invariants and introduce backward invariants which in some sense formalize the validity of prophecies. We introduce eternity variables to prove refinement relations between specifications. Section 3 contains an informal description of the serializable database interface of [18]. In Section 4, we develop our formal

specification, using an eternity variable to formalize serializability. The implementation of [18] based on locking objects is described in Section 5.

Section 6 contains the verification of this implementation. Here, the eternity variable of the specification gets its value as a limit of a history variable. A second eternity variable is introduced to prescribe the success and the order of the transactions of the database. In Section 7, we describe the verification of our proof with the first-order mechanical theorem prover NQTHM. We draw conclusions in Section 8. The new results are in Sections 2, 4, 6, and 7.

2. The formal computational model

In this section, we present the formal model, which is a semantical version of Lamport's TLA [1,13] without the syntactic restrictions and conventions. We generalize the concept of invariance and introduce a new proof rule for so-called backward invariance. We briefly present the simulation theory we proposed at MPC 2002 [7]. The paper [7] contains an error in the completeness result, which has been corrected in [8]. In the present paper we only need and prove soundness of eternity extensions. This section is rather heavy. We refer to [8] for a more balanced presentation of the theory with more examples.

We use lists to represent consecutive states during computations. The elements of a list xs are xs_i for $i \geqslant 0$. If X is a set, we write X^ω for the set of infinite lists over X. If f is a function $X \to Y$ then $f^\omega : X^\omega \to Y^\omega$ is the function lifted to infinite lists. If s is a finite list, s^ω stands for the list obtained by concatenating infinitely many copies of s.

For infinite lists xs and ys, we define xs to be a *stuttering* of ys, iff xs is obtained from ys by consecutive repetition of certain elements. For example, with X containing a, b, and c, the infinite list $(aaabbbc)^\omega$ is a stuttering of $(abbc)^\omega$. A subset P of X^ω is called a *property* [1] over X iff, whenever xs is a stuttering of ys, we have $xs \in P$ if and only if $ys \in P$.

For a subset P of X^ω, we write $\neg P$ to denote its complement (negation). We write $Suf(xs)$ to denote the set of infinite suffixes of an infinite list xs. We define $\Box P$ (always P), and $\Diamond P$ (sometime P) as the subsets of X^ω given by

$$xs \in \Box P \equiv Suf(xs) \subseteq P,$$
$$\Diamond P = \neg\Box\neg P.$$

If P is a property then $\neg P$, $\Box P$, and $\Diamond P$ are properties. For a subset U of X, we define the set $[\![U]\!]$ to consist of the infinite lists xs with initial element $xs_0 \in U$. The set $[\![U]\!]$ is always a property. For a binary relation A, regarded as a subset of $X \times X$, the subset $[\![A]\!]$ of X^ω consists of the infinite lists xs with $(xs_0, xs_1) \in A$. Usually, $[\![A]\!]$ is not a property. The set $\Box[\![A]\!]$ is a property when A is reflexive (contains the identity relation 1_X of X).

2.1. Specifications and invariants

A *specification* in the sense of [1,7,8] is a tuple $K = (X, Y, N, P)$ that consists of a state space X, a subset $Y \subseteq X$ that holds the initial states, a reflexive relation

$N \subseteq X \times X$, and a supplementary property $P \subseteq X^\omega$. It is required that P is, indeed, a property.

The elements of X are called *states*. Relation N is called the next-state relation or step relation. We define an *execution* of K to be a list xs of states for which every pair of consecutive elements belongs to N. Reflexivity of N serves to allow or eliminate stuttering: if xs is an execution, any list ys obtained from xs by repeating elements of xs and possibly removing duplicates from xs is an execution as well. An execution of K is called *initial* iff $xs_0 \in Y$. Property P is intended to express fairness and progress requirements. We define a *behaviour* of K to be an infinite and initial execution xs of K with $xs \in P$. We write $Beh(K)$ to denote the set of behaviours of K. We thus have

$$Beh(K) = [\![Y]\!] \cap \Box [\![N]\!] \cap P.$$

A state is called *reachable* iff it occurs in an initial execution. A set of states is called a *forward invariant* iff it contains all reachable states. A state is called *occurring* iff it occurs in some behaviour. A set of states is called an *invariant* iff it contains all occurring states. Since every occurring state is reachable, every forward invariant is an invariant.

The specification is called *machine closed* [1] if every finite initial execution can be extended to a behaviour (an element of $Beh(K)$). If the specification is machine closed, all reachable states are occurring and the concepts of invariant and forward invariant are equivalent, and there is no reason to distinguish between them. In this paper, however, we encounter specifications that are not machine closed. The following example provides a simple case.

Example A. Consider the program (or rather specification)

var $\mathtt{n} : Int := 0$;
do $\mathtt{n} = 0 \quad \rightarrow \quad$ *choose* $\mathtt{n} \in Int$;
[] $\mathtt{n} \neq 0 \quad \rightarrow \quad \mathtt{n} := \mathtt{n} - 2$ **od** ;
prop: infinitely often $\mathtt{n} = 6$.

This is modelled by the specification $K = (X, Y, N, P)$ with $X = \mathbb{Z}$ and $Y = \{0\}$. The step relation N consists of the pairs (n, n') with $n = 0$ or $n' = n - 2$ or $n' = n$ (to allow stuttering). The supplementary property is $P = \Box\Diamond [\![\{6\}]\!]$, which consists of the lists of natural numbers that contain infinitely many numbers 6. Informally, it is obvious that n must remain even (and nonnegative) in order to satisfy the property. Therefore, the set D of the even integers is an invariant. It is not a forward invariant since it does not contain the reachable states 3 and -1.

2.2. Proof rules for invariants

A set D of states is called a *strong invariant* (or inductive [15]) iff D contains the initial set Y and satisfies $x' \in D$ for every pair $(x, x') \in N$ with $x \in D$. It is easy to verify that a strong invariant contains all reachable states and is therefore a forward invariant.

The theory is most easily formulated in terms of sets of states, but for programming it is more convenient to use state predicates, i.e., boolean functions on the state space. We

therefore identify a state predicate Q with the corresponding set $(Q) = \{x \in X \mid Q(x)\}$. Predicate Q is called an invariant if and only if the set (Q) is an invariant.

Inspired by [15], we use the following notation to ease our calculations. Recall that $K = (X, Y, N, P)$ with $Y \subseteq X$ and $N \subseteq X \times X$. Let fst and snd be the two projection functions from $X \times X$ to X. For any set Z and a state function $g : X \to Z$, we define $g^0 = g \circ fst : N \to Z$ and $g^+ = g \circ snd : N \to Z$. By convention, the superscript ^{0} is omitted. It results that function g^+ on N stands for the value of g in the post-state whereas g itself stands for the pre-state value.

When E is some boolean function on a set W, we define $W \models E$ to mean that all elements $w \in W$ satisfy $E(w)$. As a first application, we see that a predicate Q is a strong invariant iff $Y \models Q$ and $N \models (Q \Rightarrow Q^+)$.

The classical way to prove that a family of predicates $(J_i)_i$ is a family of invariants (e.g., see [5], or [5, 3.1]) is to prove

$$\begin{array}{l} Y \models J_k \quad \text{for all } k, \\ N \models (\forall i :: J_i) \Rightarrow J_k^+ \quad \text{for all } k. \end{array}$$

Writing $J = (\forall i :: J_i)$, the first condition implies that J holds initially. The second condition implies that J is stable. It follows that J is a strong invariant, so that J and all its conjuncts J_i are forward invariants. We shall use the following easy variation of this rule.

Lemma 0. *Let D be a predicate with $Y \models D$ and $N \models (D \wedge J \Rightarrow D^+)$ for some invariant J. Then D is an invariant.*

Proof. We have to prove that $D(xs_n)$ holds for every behaviour xs and every index n. Let xs be given. We use induction on n. Since xs is a behaviour, $D(xs_0)$ holds because of $Y \models D$. We have $D(xs_n) \Rightarrow D(xs_{n+1})$ because of $J(xs_n)$ and $N \models (D \wedge J \Rightarrow D^+)$. □

The reader may also verify that, in this lemma, D is a forward invariant if J is a forward invariant.

When forward invariants are not good enough, we need a proof rule for invariants in which the supplementary property plays a role. We therefore define a predicate A to be an *attractor* iff $Beh(K) \subseteq \Box\Diamond[\![A]\!]$ or, equivalently, iff $A(xs_n)$ holds infinitely often for every behaviour xs.

Lemma 1. *Let J_0, J_1, J_2 be invariants. Let A be an attractor. Let Q be a predicate such that*

$$\begin{array}{lll} \text{(a)} & X \models & J_0 \wedge A \Rightarrow Q\,, \\ \text{(b)} & N \models & J_1 \wedge J_2^+ \wedge Q^+ \Rightarrow Q. \end{array}$$

Then Q is an invariant.

Proof. We have to prove that $Q(xs_n)$ holds for every behaviour xs and every index n. Let xs and n be given. Since A is an attractor there is $m \geqslant n$ with $A(xs_m)$. Since J_0 is

an invariant, assumption (a) implies that $Q(xs_m)$ holds. Since J_1 and J_2 are invariants, assumption (b) implies that $Q(xs_k) \Leftarrow Q(xs_{k+1})$ for every index k. Therefore $Q(xs_n)$ follows from $Q(xs_m)$ by $m-n$ backward steps. □

In the applications of Lemmas 0 and 1, we often omit the components $X \models$ and $N \models$, since the state space and step relation are supposed to be self-evident.

Example A (continued). In the setting of example A, we have $D = (Q)$ for the predicate Q given by $Q : \mathtt{n}\ \mathbf{mod}\ 2 = 0$. We use the attractor $A : \mathtt{n} = 6$. It is clear that $X \models (A \Rightarrow Q)$. Condition $N \models (Q^+ \Rightarrow Q)$ is verified in

$$(n = 0 \vee n' = n-2 \vee n' = n) \wedge n'\ \mathbf{mod}\ 2 = 0 \quad \Rightarrow \quad n\ \mathbf{mod}\ 2 = 0.$$

So, taking the invariants $J_i = true$, Lemma 1 implies that Q is an invariant.

Invariants obtained by the above lemma might be described as backward invariants, but we attach no formal meaning to this since we impose no conditions on the kind of invariance of J_0, J_1, J_2. Condition 1(b) is called *backward stability* of Q.

Remark. It may be tempting to use or propose a mixed rule that would conclude invariance of J and Q from the assumptions $Y \models J$, and $X \models (A \Rightarrow Q)$ for some attractor A, and

$$N \models (J \wedge Q \Rightarrow J^+) \wedge (J \wedge Q^+ \Rightarrow Q)\ .$$

This proposal is unsound! For example, consider the program

var n : *Nat* := 0 ;
do *true* → n := n + 1 **od** ;
prop: infinitely often n > 9 .

This program is modelled by the specification $K = (X, Y, N, P)$ with $X = \mathbb{N}$ and $Y = \{0\}$. Relation N consists of the pairs (n, n') with $n' \in \{n, n+1\}$. Property P consists of the lists of natural numbers that contain infinitely many numbers > 9. We use the attractor $A : \mathtt{n} > 9$, and the predicates $J : \mathtt{n} = 0$ and $Q : \mathtt{n} > 9$. The premises of the proposed rule hold trivially, since both $J \wedge Q$ and $J \wedge Q^+$ are identically false, but of course neither J nor Q is an invariant.

2.3. Implementation and simulation

Since we need to relate different specifications, we introduce the notations $states(K) = X$, $start(K) = Y$, $step(K) = N$, $prop(K) = P$ for a specification $K = (X, Y, N, P)$.

A specification becomes *visible* by giving a function to observe the states. In practice, the set $states(K)$ is usually a subset of a cartesian product $V \times M$, where V is spanned by so-called visible variables and M is spanned by auxiliary variables. In that case, the projection function $fst : states(K) \to V$ is used to observe the states. The auxiliary variables that span M are called *hidden*. A hidden variable that may be initialized nondeterministically and is never modified thereafter, is called an *eternity variable*. The following example shows that eternity variables can be used for specification.

Example B. Consider the program

var n : *Int* := 0 , m : *Int* ;
do *true* → n := n + 1 **od** ;
prop: eventually always n = m .

This is modelled by the specification $K=(X,Y,N,P)$ with $X=\mathbb{Z}\times\mathbb{Z}$ and $Y=\{0\}\times\mathbb{Z}$. The step relation N consists of the pairs $((n,m),(n',m))$ with $n'\in\{n,n+1\}$. The supplementary property is $P=\Diamond\Box[\![\,\mathtt{n}=\mathtt{m}\,]\!]$. This expresses that the program terminates in a state with $\mathtt{n}=\mathtt{m}$. Let us assume that m is a hidden variable. Then, it is an eternity variable. It serves here to specify termination.

It is useful to regard visibility in a more abstract way. A *visible specification* is defined as a pair (K,g) consisting of a specification K and a function g on $states(K)$ that serves to observe the states. The *visible behaviours* of (K,g) are the infinite lists $g^\omega(xs)$ with $xs\in Beh(K)$. Let (K,g) and (L,h) be visible specifications with g and h mapping to the same set. We then define (K,g) to *implement* (L,h) iff every visible behaviour of (K,g) is a visible behaviour of (L,h), i.e., iff for every $xs\in Beh(K)$ there exists $ys\in Beh(L)$ with $g^\omega(xs)=h^\omega(ys)$.

This concept of implementation is inspired by the one of [1], but it is slightly stricter: in [1] it is allowed that a visible behaviour of (K,g) only becomes a visible behaviour of (L,h) after adding stutterings. We use simulations to prove implementation relations between visible specifications. There are several versions of different generality.

Let K and L be specifications. A function $f: states(K)\to states(L)$ is called a *refinement mapping* [1] from K to L iff $f(x)\in start(L)$ for every $x\in start(K)$, and $(f(x),f(x'))\in step(L)$ for every pair $(x,x')\in step(K)$, and $f^\omega(xs)\in prop(L)$ for every $xs\in Beh(K)$. Refinement mappings form the simplest way to compare different specifications.

We define a binary relation F between $states(K)$ and $states(L)$ to be a *simulation* $K\rightarrowtriangle L$ [7,8] iff, for every behaviour $xs\in Beh(K)$, there exists a behaviour $ys\in Beh(L)$ with $(xs_n,ys_n)\in F$ for all n. Simulations are relevant for implementation because of the following completeness result.

Theorem 2 (Hesselink [8]). *Let (K,g) and (L,h) be visible specifications with g and h mapping to the same set. Then (K,g) implements (L,h) if and only if there is a simulation $F: K\rightarrowtriangle L$ with $F\subseteq\{(x,y)\mid g(x)=h(y)\}$.*

We only need the trivial part of this result, viz., that (K,g) implements (L,h) when there is a simulation $F: K\rightarrowtriangle L$ with $F\subseteq\{(x,y)\mid g(x)=h(y)\}$. See [8] for the proof of the converse implication.

The easiest examples of simulations come from subspaces. Indeed, consider a specification $K=(X,Y,N,P)$ and a set of states $R\subseteq X$. The *subspace restriction* K_R of K to R is defined as the tuple $(R,Y\cap R,N\cap R^2,P\cap R^\omega)$. It is easy to verify that K_R is a specification and that the identity function 1_R is a refinement mapping from K_R to K. It is also easy to verify that the converse relation 1_R is a simulation $K\rightarrowtriangle K_R$ if and

only if R is an invariant. In that case, the simulation $K \rightarrowtriangle K_R$ is called the associated *invariant restriction*.

2.4. Variable extensions

In programming practice, most simulations occur by extending some program with auxiliary variables. Formally, a specification L is called an extension (or variable extension) of specification K with a variable of a type M iff $states(L)$ is a subset of the cartesian product $states(K) \times M$. The second component of the states of L is then regarded as the variable added. The projection function $fst : states(L) \rightarrow states(K)$ is often a refinement mapping, but this is usually irrelevant. The extension is called a *refinement extension* iff the converse relation $cvf = cv(fst)$ is a simulation $K \rightarrowtriangle L$.

The extension is called a *history extension* (or an extension with a history variable) iff

(H0) For every $x \in start(K)$ there is $m \in M$ with $(x, m) \in start(L)$.
(H1) For every pair $(x, m) \in states(L)$ and x' with $(x, x') \in step(K)$ there is $m' \in M$ with $((x, m), (x', m')) \in step(L)$.
(H2) $prop(L)$ consists of the lists ys with $fst^{\omega}(ys) \in prop(K)$.

It is easy to see that, if L is a history extension of K, relation cvf is a simulation $K \rightarrowtriangle L$, e.g., [8]. So, every history extension is a refinement extension. History extensions go back to [17] where history variables are called auxiliary variables.

Eternity extensions, introduced in [7,8], are another kind of variable extensions. The starting point is the *trivial* history extension $K\xi M$ of K by M, in which the variable added is never modified and does not interact:

$$
\begin{array}{l}
states(K\xi M) = states(K) \times M \ , \\
start(K\xi M) = start(K) \times M \ , \\
((x, m), (x', m')) \in step(K\xi M) \quad \equiv \quad (x, x') \in step(K) \wedge m = m' \ , \\
ys \in prop(K\xi M) \quad \equiv \quad fst^{\omega}(ys) \in prop(K) \ .
\end{array}
$$

So, the state space is extended with an unknown nondeterministic constant $m \in M$, in other words with an *eternity variable* m.

A binary relation R between $states(K)$ and M (i.e. a subset of $states(K\xi M)$) is called a *behaviour restriction* between K and M iff, for every behaviour xs of K, there exists an $m \in M$ with $(xs_i, m) \in R$ for all indices i:

$$
\text{(BR)} \qquad xs \in Beh(K) \quad \Rightarrow \quad (\exists\, m :: (\forall i :: (xs_i, m) \in R)) \ .
$$

If R is a behaviour restriction between K and M, we define the corresponding *eternity extension* as the subspace restriction $W = (K\xi M)_R$ of the trivial history extension $K\xi M$. So, we have $states(W) = R$, and $start(W)$, $step(W)$, and $prop(W)$ are the natural restrictions of $start(K\xi M)$, $step(K\xi M)$, and $prop(K\xi M)$ to R or its liftings.

The soundness of eternity extensions is expressed in the following easy result:

Lemma 3 (Hesselink [8]). *Let R be a behaviour restriction. Then relation cvf is a simulation $K \rightarrowtriangle W$.*

Proof. Let $xs \in Beh(K)$. We have to construct $ys \in Beh(W)$ with $(xs, ys) \in cvf^{\omega}$. By (BR), we can choose m with $(xs_i, m) \in R$ for all i. Then we define $ys_i = (xs_i, m)$. A trivial verification shows that the list ys constructed in this way is a behaviour of W with $(xs, ys) \in cvf^{\omega}$. This proves that cvf is a simulation. □

We refer to [8] for a presentation of the theory with more examples. In [8], it is also proved (using a slightly different terminology) that the combination of history extensions and eternity extensions is in a certain sense semantically complete: every simulation that also "preserves quiescence" is a composition of a history extension, an eternity extension and a refinement mapping.

In Section 4 below, we use an eternity variable in a specification. Backward invariants and eternity extensions will be used in Section 6.

3. An informal description of a database interface

We come back to the database interface mentioned in the introduction. The following description is extracted from [18], see also [4,10,11].

A database is a system of *objects* that can be read and written by a collection of client processes. Each client process performs a sequence of transactions, where a transaction consists of a sequence of reads and writes to database objects. Clients may concurrently execute transactions. Transactions may be aborted either by the client or by the database. The result of aborting a transaction is as if none of the reads and writes of the transaction were executed. Serializability means that the values returned by all the read operations from successful transactions are ones that could be obtained by executing these transactions in some sequential order: an order in which all reads and writes of one transaction are performed before any operation of the next transaction is performed.

We also impose a weak restriction on the serialization order in terms of the temporal order of the transactions: we require that for every behaviour it is possible to insert a unique serialization event into every successful transaction such that the serialization order corresponds to the temporal order of serialization events. The paper [18] does not mention such a requirement and thus allows the implementor more freedom of implementation. We don't know whether this was intended.

A client accesses the database by the following procedure calls. It must wait for the return of a call before issuing another call. Different clients may issue concurrent calls.

Begin-T(): *key* or *failed*

Called to initiate a transaction. The value returned is a key to identify the transaction in the other calls. The call may only *fail* because of some lack of resources.

Read(*key*, *object*): *value* or *abort*

This returns the current value of the object in the database, unless the transaction is aborted.

Write(*key*, *object*, *value*): *ok* or *abort*

Writes *value* to *object* and returns *ok*, unless the transaction is aborted.

End-T(*key*): *ok* or *abort*

Ends the transaction with *ok*, unless it is aborted by the database.

Abort(*key*)

Aborts the transaction; this is always successful.

We allow a key of a transaction that has been terminated or aborted to be reused. The last four calls only occur with a key of an active transaction, i.e., one that has begun more recently than it has been terminated or aborted (if ever).

The procedures *Read*, *Write* and *End-T* may only be aborted by the database if the transaction accesses an object that is concurrently accessed in some other transaction. It is assumed that, if a transaction is not aborted, the client will terminate it by a call to *End-T* after a finite number of calls. Under this assumption, the database guarantees that control eventually returns from each procedure call.

Remark. The successful transactions must be serialized while retaining the order of the actions within each transaction. The order of actions from different transactions may be changed. Consider, for example, the following scenario where the database contains an integer object x, initially $\mathrm{x} = 17$. Client A has a private variable a and performs the transaction that consists of $a := \mathrm{x}$ followed by $\mathrm{x} := a + 1$. Client B performs the transaction that consists of the single write action $\mathrm{x} := 8$. When the transactions of A and B overlap, the transactions may be aborted. When both transactions succeed, the pair (a, x) in the final state can be either $(17, 8)$ or $(8, 9)$.

4. The specification of the database

In this section we develop a formal specification. We begin with the introduction of the main state variables and a discussion of the flow of control in Section 4.1. We model the clients of the database as a nondeterministic environment in Section 4.2. The database is specified by means of an abstract program in 4.3. We convert this program in a relational specification in 4.4. We justify the specification in Section 4.5.

4.1. Main state variables and global control

The database to be implemented has an environment which we do not control. This environment repeatedly submits invocations to the database that belong to some key.

For simplicity and uniformity, we assume that *Begin-T* just chooses an unused key, independently of the database system. We assume that *Begin-T* is called only when such an unused key is available. It therefore never leads to immediate abort.

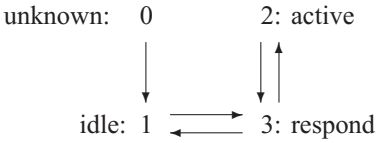
Even though keys can be reused, we allow the set of available keys to be infinite. A key becomes *known* when the environment starts using it. Initially, there are no known keys. The set of known keys can only grow, but it remains finite.

The invocations for reading and writing are combined and potentially generalized in a set *Inv* of *ordinary* invocations. We use additional symbols B, E, A for the special invocations *Begin-T*, *End-T*, *Abort*. Similarly, *Res* is the set of results of ordinary invocations. We introduce for each key k the variables

$$\begin{array}{l} turn.k : \{0,1,2,3\} \ ; \\ inv.k : \ Inv \cup \{\mathtt{B},\mathtt{E},\mathtt{A}\} \ ; \\ res.k : \ Res \cup \{\mathtt{B},\mathtt{E},\mathtt{A}\} \ ; \\ accessed.k : \ \textbf{set of } Obj \ . \end{array}$$

The variables associated to key k are called the private variables of k. Indeed, we can regard key k as a process identifier. To distinguish private variables from shared ones, we use slanted font for private variables and typewriter font for shared variables.

The private variable *turn.k* holds the status of the key; $turn.k = 0$ means that the key is (still) unknown; $turn.k = 1$ holds for keys that are known but not in a transaction; $turn.k \in \{2,3\}$ means that the key is involved in a transaction, with $turn.k = 2$ when the environment may submit a new invocation and $turn.k = 3$ when the database has to respond to a current invocation. So, the database can modify *turn.k* only when it equals 3 and the environment can modify it only when it differs from 3. Once a key k is known, i.e., $turn.k \neq 0$, the environment and the database will never reset *turn.k* to zero. The transitions of *turn* are described in the following diagram.



The database must be fair: it has the obligation eventually to treat every invocation, i.e., to enable the environment infinitely often by setting *turn.k* to a value $\neq 3$. This is formalized in the condition that *turn.k* always eventually differs from 3, as expressed in the temporal formula

Fdat: $\quad \Box\Diamond \ (turn.k \neq 3) \ .$

The private variable *inv.k* holds the current invocation at key k, whereas *res.k* holds the latest result at this key. The private variable *accessed.k* holds the set of objects accessed in the current transaction with key k. We use the convention that $accessed.k = \emptyset$ for unknown keys k. The visible variables are *inv.k*, *res.k*, *turn.k*. The variables *accessed.k* and all variables introduced below only serve in the specification.

4.2. Modelling the environment and its expectations

In order to specify the database we model the clients by a very nondeterministic environment that may submit new invocations when $turn \neq 3$. Such a new invocation is B when there is no current transaction and $\neq$B otherwise. The specification uses a number of auxiliary variables (both shared and private) that need not be implemented but only serve to constrain the visible behaviours of the system.

In order to ensure that all transactions terminate, we specify that, at the start of a transaction of key k, the environment chooses an upper bound *evf* for the number of nontrivial invocations in the transaction. Of course, the database will not be allowed to inspect or modify *evf*.

Every key gets a private integer variable *nr* to hold the number of the current successful transaction in the serialization ordering. It gets its value at some moment during the transaction. This moment is called the serialization event. By convention $nr.k = 0$ when the current transaction of key k is not successful or before the serialization event. Every key gets a private boolean variable *sysAb* to indicate that the database is allowed to abort the current transaction. We thus introduce the private variables

$$evf, nr : \ Int \ ;$$
$$sysAb : Boolean \ .$$

Since serialization may occur at any time during a transaction, even before the database has answered the invocation B, the environment at k initializes $nr.k := 0$ when it creates invocation B. Variable *sysAb* is initialized at this point for the same reason.

We write $|inv|$ for the set of objects to be accessed in invocation *inv*. In the setting of Section 3, the sets $|inv|$ contain never more than one object. The component command for the environment at key k is given by:

$Env.k$: **whenever** $turn = 0 \ \rightarrow \ res := \mathtt{A} \ ; \ turn := 1$
[] $turn = 1 \ \rightarrow$
$choose \ evf > 0 \ ; \ nr := 0 \ ; \ sysAb := false \ ;$
$inv := \mathtt{B} \ ; \ turn := 3$
[] $turn = 2 \ \rightarrow$
$evf := evf - 1 \ ;$
$choose \ inv \in Inv \cup \{\mathtt{E}, \mathtt{A}\} \ with \ (evf \leqslant 0 \Rightarrow inv \in \{\mathtt{E}, \mathtt{A}\}) \ ;$
$accessed := accessed \cup |inv| \ ; \ turn := 3$
end .

In this guarded command notation, each alternative is regarded as an atomic step that can be taken whenever its guard holds. The command expresses a part of the next-state relation N in the sense of the formal model of Section 2. We use the keyword **whenever** to emphasize that these steps can be repeated infinitely often. When all guards are false, the component has to wait until (possibly) some other component makes some guard true again. We do not use the **do od** notation, since that would suggest termination when all guards are false.

Command *Env.k* expresses the steps the environment at key k can take, when $turn.k = 0$, 1 or 2. When $turn.k = 3$, the environment at key k is disabled and has to wait for the database to make some guard true.

In these component programs, we suppress the index k for all private variables of key k. When an unknown key is initialized, its field *res* is set to `A` to indicate that it has no previous transaction that delivered meaningful results.

The environment program *Env* is the parallel composition $(||_k Env.k)$ of the environment programs of all keys. Here, parallel composition is defined as the union (disjunction) of the corresponding next-state relations, i.e., by interleaving semantics. The environment *Env* is not subject to implementation.

4.3. Modelling the abstract database

In this section, we develop the logical database as an abstract program, as a stage towards the relational specification in Section 4.4. We abstract from the objects in the database and their individual values as much as possible. We introduce a set *DbVal* for the set of states of the total database with the element $\mathtt{db0} \in DbVal$ as the initial state. We assume that calls are specified by a function

$$DbF : Inv \times DbVal \quad \rightarrow \quad Pow(Res \times DbVal) \ ,$$

where $(r, w) \in DbF(i, v)$ means that invocation i in database state v may return result r and transform the database state into w. We assume that the set $DbF(i, v)$ is nonempty for every pair i, v. Since we want the specification not to impose any restrictions on the results of failing transactions, we model the database state of a failing transaction by $\perp \notin DbVal$ and extend function DbF to $DbVal \cup \{\perp\}$ by defining $DbF(i, \perp) = \{(r, \perp) \mid r \in Res\}$ for all $i \in Inv$.

We turn to the development of an abstract database program *Adb*. The implementor of the database has to provide a concrete program *Cdb* such that every execution of the parallel composition $(Env||Cbd)$, when projected to the visible elements *inv.k*, *res.k*, and *turn.k* for all keys k, is also the projection of some execution of $(Env||Adb)$ for the abstract program *Adb*. Notice that both parallel compositions are subject to the supplementary property *Fdat*.

We specify the responses of the database as liberally as possible and yet enforce serializability. We do this by prescribing a highly nondeterministic next-state command, which leads to a deadlocked key when some choices do not fit. The supplementary property *Fdat* will eliminate executions with deadlocked keys. This means that the abstract specification will not be machine closed, i.e., that there are finite executions that cannot be extended to acceptable behaviour since *Fdat* cannot be satisfied.

In order to specify serializability, i.e., to enforce some conceptual serialization, we introduce a shared counter `gnr` and a shared array `etMem` declared by

$$\mathtt{gnr} : \ Nat := 0 \ ;$$
$$\mathtt{etMem} : \ \textbf{array} \ \ Nat \ \ \textbf{of} \ \ DbVal \ .$$

It is the intention that successful transaction n transforms $\mathtt{etMem}[n-1]$ into $\mathtt{etMem}[n]$ and that `gnr` holds a number of a recent successful transaction. We number the successful transactions from 1 and assume that $\mathtt{etMem}[0]$ holds the initial state `db0` of the database. The other elements of `etMem` are initialized nondeterministically. More precisely, the specification allows all behaviours for which there exists a consistent value

for etMem. Therefore, etMem is an eternity variable. The identifier etMem is chosen to reflect this fact. If a behaviour contains only a finite number of successful transactions, the unused tail of etMem is irrelevant (it may stutter, but need not do so).

Every key gets private variables *start* and *val* to hold the initial and current database states during successful transactions. By convention *start* and *val* can be $\perp$ during failing transactions. Comparison of *start.k* and *val.k* with $\texttt{etMem}[nr.k-1]$ and $\texttt{etMem}[nr.k]$ for the acting key k will serve to enforce serialization. We thus introduce the private variables

$$start,\ val:\ \ DbVal \cup \{\perp\}.$$

We introduce a state function *conflict* to denote that key k is concurrently accessing a common object with some other key (m):

$$conflict(k) \quad = \quad (\exists\, m :: m \neq k \ \wedge\ accessed.m \cap accessed.k \neq \emptyset).$$

When $conflict(k)$ holds, the database is allowed to abort k's current transaction. This can be recorded in the private boolean variable *sysAb.k*.

The next-state command *Adb* of the abstract database program is the parallel composition $(||_k Adb.k)$ of the component commands for all keys:

Adb.k :

(B) **whenever** $turn = 3 \ \wedge\ inv = \texttt{B} \ \rightarrow$
  choose $val \in DbVal \cup \{\perp\}$; $start := val$;
  $res := \texttt{B}$; $turn := 2$

(I) [] $turn = 3 \ \wedge\ inv \in Inv \ \rightarrow$
  choose $(res, val) \in DbF(inv, val)$; $turn := 2$

(A) [] $turn = 3 \ \wedge\ nr = 0 \ \wedge\ (inv = \texttt{A} \vee sysAb) \ \rightarrow$
  $accessed := \emptyset$; $res := \texttt{A}$; $turn := 1$

(E) [] $turn = 3 \ \wedge\ inv = \texttt{E} \ \wedge\ nr > 0$
  $\wedge\ start = \texttt{etMem}[nr-1] \ \wedge\ val = \texttt{etMem}[nr] \ \rightarrow$
  $accessed := \emptyset$; $res := \texttt{E}$; $turn := 1$

(S) [] $turn \in \{2,3\} \ \wedge\ nr = 0 \ \rightarrow$
  $\texttt{gnr} := \texttt{gnr} + 1$; $nr := \texttt{gnr}$

(C) [] $conflict(k) \ \rightarrow\ sysAb := true$

end .

As before, we omit references to key k for all private variables. Note that the guards are not mutually exclusive. Any alternative with a true guard can be chosen as a single atomic step.

The first alternative (B) is the response of the database to a beginning invocation. This starts with the guess whether the beginning transaction will be successful and, if so, of the initial database state of the new transaction. This initial value is copied into *start*. Furthermore, result value B is generated and the environment is enabled by setting *turn* to 2.

The second alternative (I) is the ordinary invocation. If $val = \perp$, the results are nondeterministic and *val* remains $\perp$ because of the definition of $DbF(i, \perp)$.

Alternative (A) allows a failing transaction to abort either upon the client's request or because of concurrent object access during the transaction. It can only happen when the current transaction has not yet been serialized ($nr = 0$). Alternative (E) is the conclusion of a successful transaction together with tests of the initial and final state of that transaction. It can only occur after serialization, i.e. with $nr > 0$.

Alternative (S) is the serialization event: the choice of a sequence number during a presumably successful transaction. Alternative (C) is the observation that the database is allowed to abort the current transaction. The variable *sysAb* is introduced to allow some delay in the abortion. In fact, it may happen that $conflict(k)$ only holds while $turn.k = 2$; then $sysAb.k$ can be set, so that the database is allowed to abort at the next turn. Notice that the guards of (S) and (C) both imply that the key is in a transaction.

4.4. The relational specification

Command *Adb* specifies the database, but does not provide a convenient test to decide whether or not some proposed implementation is correct. We therefore replace *Adb* by a relational specification *STEP*. The above development of *Adb* only serves as a heuristic justification of *STEP*.

Command *Adb* describes the new values of certain variables, while other variables have to remain constant since they are not mentioned. At this point, *Adb* as developed above is over-specific. So, for the sake of abstractness, we note that the value of $res.k$ is irrelevant while $turn.k \in \{0, 3\}$. Similarly, the transaction variables $start.k$, $val.k$, $nr.k$, and $sysAb.k$ are irrelevant while $turn.k \in \{0, 1\}$. To allow for maximal flexibility, we can therefore add an arbitrary nondeterministic choice of a variable whenever *turn* gets or has a value where that variable is irrelevant.

In the relational specification, we use the following conventions. Recall from Section 2.2 that, for a variable v, we use v for the value of v in the pre-state and v^+ for its value in the post-state [15]. Inspired by the specification language Z, we write Ξ followed by a list of variables v for the conjunction of equalities $v = v^+$. As before, we omit the index k for the private variables of key k. We use Lamport's TLA convention to add an initial $\wedge$ operator to a list of conjunctions separated by newlines.

As noted in [2], we have to distinguish between actions of the environment and actions of the database. For this purpose, we introduce a shared variable `actor` which is set to `ENV` by every action of the environment and to `DB` by every action of the database. We regard these modifications of `actor` as part of the formal setting. So, they need not be included in the specifications.

Since the environment is given and not subject to implementation, we develop a next-state relation *STEP* for actions of the database, which allows arbitrary actions of the environment.

$$STEP \quad \equiv \quad (\texttt{actor}^+ = \texttt{DB} \quad \Rightarrow \quad (\exists\, k :: adb.k)) \ ,$$

where $adb.k$ is the next-state relation for key k. The antecedent $\texttt{actor}^+ = \texttt{DB}$ means that the step is taken by the database. Relation $adb.k$ only allows modifications of the

shared variables and the private variables of key k. It is specified by

$$
\begin{array}{ll}
& adb.k \quad \equiv \\
\wedge & \Xi(\mathtt{etMem}, evf, inv) \\
\wedge & (SpB.k \vee SpI.k \vee SpA.k \vee SpE.k \vee SpS.k \vee SpC.k) \ .
\end{array}
$$

Here, we require that the database leaves etMem, *evf*, and *inv* unchanged since etMem is an eternity variable and the changes of *evf* and *inv* are left to the environment. Relation *adb.k* is further subdivided into relations *SpB*, *SpI*, *SpA*, *SpE*, *SpS*, and *SpC*, which represent the alternatives (B), (I), (A), (E), (S), (C) of command *Adb.k*, respectively. The begin (B) of a transaction is characterized by

$$
\begin{array}{ll}
& SpB.k \quad \equiv \\
\wedge & turn = 3 \ \wedge \ turn^+ = 2 \ \wedge \ \Xi(\mathtt{gnr}, accessed, nr, sysAb) \\
\wedge & inv = \mathtt{B} = res^+ \ \wedge \ start^+ = val^+ \ .
\end{array}
$$

An ordinary invocation (I) is characterized by

$$
\begin{array}{ll}
& SpI.k \quad \equiv \\
\wedge & turn = 3 \ \wedge \ turn^+ = 2 \ \wedge \ \Xi(\mathtt{gnr}, accessed, nr, sysAb, start) \\
\wedge & inv \in Inv \ \wedge \ (res^+, val^+) \in DbF(inv, val) \ .
\end{array}
$$

Abortion (A) is characterized by

$$
\begin{array}{ll}
& SpA.k \quad \equiv \\
\wedge & turn = 3 \ \wedge \ turn^+ = 1 \ \wedge \ nr = 0 \ \wedge \ \Xi(\mathtt{gnr}) \\
\wedge & (inv = \mathtt{A} \vee sysAb) \ \wedge \ res^+ = \mathtt{A} \ \wedge \ accessed^+ = \emptyset \ .
\end{array}
$$

Successful termination (E) is characterized by

$$
\begin{array}{ll}
& SpE.k \quad \equiv \\
\wedge & turn = 3 \ \wedge \ turn^+ = 1 \ \wedge \ nr > 0 \ \wedge \ \Xi(\mathtt{gnr}) \\
\wedge & start = \mathtt{etMem}[nr-1] \ \wedge \ val = \mathtt{etMem}[nr] \\
\wedge & inv = \mathtt{E} = res^+ \ \wedge \ accessed^+ = \emptyset \ .
\end{array}
$$

Serialization (S) is characterized by

$$
\begin{array}{ll}
& SpS.k \quad \equiv \\
\wedge & turn \in \{2,3\} \ \wedge \ nr = 0 \ \wedge \ \mathtt{gnr}^+ = nr^+ = \mathtt{gnr} + 1 \\
\wedge & (turn = 3 \vee \Xi(res)) \\
\wedge & \Xi(turn, accessed, start, val, sysAb) \ .
\end{array}
$$

Alternative (C) of *Adb.k* is combined with the possibilities for silent steps in

$$
\begin{array}{ll}
& SpC.k \quad \equiv \\
\wedge & ((\neg sysAb \ \wedge \ conflict(k)) \vee \Xi(sysAb)) \\
\wedge & (turn = 3 \vee \Xi(res)) \\
\wedge & \Xi(\mathtt{gnr}, turn, accessed, start, val, nr) \ .
\end{array}
$$

Both *SpS* and *SpC* allow the database to modify *res* gradually while $turn = 3$.

An implementation *Cdb* of the database satisfies the specification if and only if it can be extended with actions on the ghost variables gnr, etc., such that every behaviour

of its parallel composition $(Env||Cdb)$ with program Env, that starts in an initial state with $\mathtt{gnr} = 0$ and $\mathtt{etMem}[0] = \mathtt{db0}$ and $turn.k = 0$ for all k, satisfies $\Box[\![STEP]\!]$ and the fairness property $Fdat$.

4.5. Justification of the formal specification

This section is devoted to the question whether the specification of Section 4.4 indeed specifies the informal description of Section 3. In particular, whether it specifies serializability and does not impose undue restrictions on implementations. Of course, the correspondence of a formal specification with an informal description cannot be proved formally. So we cannot give a proof, we can only give convincing arguments.

We use the term *history* for a sequence of actions in terms of the informal description. We first show that every acceptable history corresponds to a behaviour of the specification. Secondly, we indicate why a behaviour of the specification corresponds to a history that is acceptable for the informal description.

4.5.1. Informal satisfies formal

Firstly, assume that we have an acceptable history. We have to show that this history can occur in the specification. By assumption, the initial state of the history fits the initial state of the specification. The history is a finite or infinite system of terminating transactions. The failing transactions of the history correspond to transactions of the specification where val and $start$ are chosen as $\perp$. The specification allows that they have all kinds of intermediate results. The successful transactions can be serialized in some order.

Let $m \in \mathbb{N} \cup \{\infty\}$ be the number of successful transactions of the history. Following [14, Section 13.1], we interpret serializability to mean that these transactions can be numbered T_n with $1 \leqslant n \leqslant m$ such that the history corresponds to sequential execution of $T_1; T_2; T_3; \ldots$ and that, for every n, successful transaction T_n contains a unique serialization event $(\mathtt{S}, n)$ that occurs after its beginning event and before its termination event, in such a way that $(\mathtt{S}, i)$ happens before $(\mathtt{S}, j)$ whenever $i < j$.

It follows that there exists a sequence of states $(v_n)_n$ of the database such that, for all $n \leqslant m$, state v_{n-1} is the initial state of T_n and v_n is the final state of T_n. Clearly, $v_0 = \mathtt{db0}$, the initial state of the database.

We now describe the construction of a matching execution of $STEP$. First, we initialize array $\mathtt{etMem}$ by choosing $\mathtt{etMem}[n] = v_n$ for all $n \leqslant m$. We proceed by inspecting the history. Whenever a key k begins a failing transaction in the history, the new values $start := \perp$ and $val := \perp$ are chosen in $SpB.k$. Key k then continues the transaction with steps of the form $SpI.k$, which can return arbitrary meaningless results, until it fails by execution of $SpA.k$ with $res^+ = \mathtt{A}$. If it aborts because of a conflict during the transaction, at some point $sysAb$ is made true in alternative $SpC.k$.

Whenever a key k starts a successful transaction, say with number n in the history, $SpB.k$ chooses values $start.k$ and $val.k$ equal to v_{n-1}. The key then proceeds with the transaction, possibly interleaved by transactions at other keys. All intermediate invocations of key k lead to results res and modify the global state according to relation DbF, as specified in $SpI.k$.

At the moment of the serialization, we take the step *SpS.k* for the key k that is executing transaction T_n. Using induction, we have $\texttt{gnr} = n-1$ as a precondition of this step and, in this step, key k sets `gnr` and *nr.k* equal to n. After the assignment to *nr* and due to the choice of $\texttt{etMem}[n] = v_n$, the terminating invocation of the transaction in the history indeed has a corresponding step *SpE.k* in the specification. This concludes our argument that the informal description satisfies the specification.

4.5.2. Formal satisfies informal

Conversely, consider a behaviour of specification *STEP*. Every invocation by the environment is eventually answered by the database because of property *Fdat*. Relation *adb* shows that the invocations `E` and `A` are answered by `E` or `A`. Using decrementation of *evf*, it follows that every transaction of the behaviour terminates.

The transactions with final result `E` can clearly be pasted together according to the sequence numbers to form an acceptable sequential history of the database with `etMem` as sequence of global database states. Since they started with database values $val.k \neq \bot$, they yielded results consistent with sequence `etMem`. These transactions moreover have chosen consecutive sequence numbers *nr* during their lifetime. This proves serializability.

Finally, a transaction of key k is only aborted by the database if the last invocation was `A` or if *sysAb.k* holds which implies that the transaction accessed an object that was also accessed concurrently by another key.

4.5.3. Formal deadlock must be avoided

As announced earlier, the specification is not machine closed: there are finite executions that cannot be extended to acceptable behaviour since fairness property *Fdat* cannot be satisfied. We need not be concerned about this. In fact, by specifying *Fdat*, we force the implementor to avoid such executions.

More explicitly, a state where $turn.k = 3$ holds, has a path to a state with $turn.k \neq 3$ only if it satisfies

$$\begin{aligned} &inv.k = \texttt{A} \quad \Rightarrow \quad nr.k = 0 \ , \\ &inv.k = \texttt{E} \ \wedge \ nr.k > 0 \\ &\quad \Rightarrow \quad start.k = \texttt{etMem}[nr.k - 1] \ \wedge \ val.k = \texttt{etMem}[nr.k] \ . \end{aligned}$$

Indeed, if either of these predicates is false, the process of key k is blocked forever, which implies violation of property *Fdat*. It follows that, in the proof of correctness of any implementation, the choice of the ghost variable *start* requires prescience (i.e. knowledge of a later invocation). In Section 6, we show how this can be done.

5. Implementation by locking objects

The implementation proposed in [18] is based on locking objects. So, now the database consists of a set of objects *Obj* with values in a certain set *Value*.

$$DbVal \quad = \quad \textbf{array} \quad Obj \quad \textbf{of} \quad Value$$

We assume that an invocation consists of an object and some command concerning the object, that the result of the invocation only depends on the command and the value of the object, and that it can only modify this value. So, there is a set *ObInv* to specify the commands at a given object and the set *Inv* introduced in 4.1 is henceforth the set of pairs $Inv = Obj \times ObInv$. We assume the responses of the database at all objects to be specified by a function

$$ObN : ObInv \times Value \quad \rightarrow \quad Pow(Res \times Value) \ ,$$

where $(r,t) \in ObN(i,u)$ means that command i for an object with value u may return result r and change the value of the object into t. We assume that $ObN(i,u)$ is always nonempty and that the global next-state function *DbF* is expressed in terms of *ObN* by

$$\begin{array}{l}(r,w) \in DbF((o,i),v) \quad \equiv \\ \quad (r,w[o]) \in ObN(i,v[o]) \ \wedge \ (\forall\ p : p \neq o : v[p] = w[p]) \ .\end{array}$$

The set of objects accessed in invocation $(o,i) \in Inv$ is naturally defined by $|(o,i)| = \{o\}$. The invocations $inv \in \{\mathtt{B}, \mathtt{E}, \mathtt{A}\}$ access no objects and have $|inv| = \emptyset$. The set *Value* has a default element `null` and the initial database value `db0` satisfies $\mathtt{db0}[o] = \mathtt{null}$ for all objects o. These assumptions are more specific than the general setting of Section 4, but general enough for the setting of [18] as exposed in Section 3. Since every invocation can modify at most one value in the database, there are always at most finitely many objects with values $\neq \mathtt{null}$, even though the set *Obj* is allowed to be infinite.

According to [18], the implementation processes interact with the physical database through the following procedures.

procedure *Acquire-L*(*ob* : *Obj*) : *Boolean* .

The lock is granted iff the procedure returns *true*; otherwise it is rejected. The lock of one object is never granted to more than one key at the same time. It becomes available again when the owner process calls:

procedure *Release-L*(*ob* : *Obj*) .

A call of *Release-L* may be issued for an object only after the lock has been granted for that object, in which case the call will eventually return. It is guaranteed that, if every granted lock is eventually released, every call to *Acquire-L* eventually returns.

We formally specify *Acquire-L* and *Release-L* by means of a shared variable

`owner` : **array** *Obj* **of** $Key \cup \{\bot\}$,

with the initial values $\mathtt{owner}[o] = \bot$ for all objects o.

Apart from finite waiting, the meaning of *Acquire-L* is captured by

```
procedure Acquire-L(ob) : Boolean =
  ⟨ if owner[ob] = ⊥ then
      owner[ob] := self ; return true
    else return false fi ⟩
end .
```

Here, *self* refers to the acting key. The angled brackets mean that the enclosed command is executed atomically. Note that *Acquire-L* allows the database to wait for some time when the object is unavailable. Indeed, atomicity does not require immediate execution.

We model that a lock can only be released by its owner by defining

procedure *Release-L*(*ob*) =
  ⟨ **assert**(owner[*ob*] = *self*) ; owner[*ob*] := ⊥ ⟩
end .

The **assert** means that the programmer must prove that the argument holds.

We use these primitives in the following implementation of the database. The physical database is declared by

db : *DbVal* := db0 .

The concrete program for the database for key *k* is given by

Cdb.k : **whenever** *turn* = 3 →
  if *inv* = B → *beginTrans*()
  [] *inv* ∈ *Inv* → **if** *owns*() **then** *handle*() **else** *aborting*() **fi**
  [] *inv* = A → *aborting*()
  [] *inv* = E → *endTrans*()
  fi
end.

The simplest auxiliary procedure is

procedure *beginTrans* () =
  res := B ; *turn* := 2
end .

For every key, we declare the private variables

ownset : **set of** *Obj* := ∅ ;
pridb : *DbVal* .

We use *ownset* to hold the objects for which the key holds the locks. The current value of such an object *ob* is kept in *pridb*[*ob*] as long as the lock is kept.

Procedure *owns* serves to verify or guarantee that this key has exclusive access to the object of the invocation. Indeed, since *Inv* is a cartesian product, we assume that *inv* = (*iob*, *iv*) whenever *inv* ∈ *Inv*. In other words, *iob* and *iv* are aliases of the two components of *inv*.

procedure *owns* () : *Boolean* =
  if *iob* ∈ *ownset* **then return** *true*
  elsif *Acquire-L*(*iob*) **then**
    ownset := *ownset* ∪ {*iob*} ; *pridb*[*iob*] := db[*iob*] ;
    return *true* ;
  else return *false* **fi**
end .

Procedure *handle* works on the private copies:

procedure *handle* ( ) =
  choose $(res, w) \in ObN(iv, pridb[iob])$;
  $pridb[iob] := w$; $turn := 2$
end .

Procedure *endTrans* commits the modifications to the database:

procedure *endTrans* ( ) =
  for all $o \in ownset$ **do** $\mathtt{db}[o] := pridb\ [o]$ **od** ;
  releaseAll( ) ; $res := \mathtt{E}$; $turn := 1$
end .

Here we use a new procedure *releaseAll* to release all locks that the key owns, to preserve the invariant concerning *ownset*, and to transfer control to the environment:

procedure *releaseAll* ( ) =
  for all $o \in ownset$ **do** *Release-L*(o) **od** ;
  $ownset := \emptyset$;
end .

This procedure is also used when aborting:

procedure *aborting* ( ) =
  releaseAll( ) ; $res := \mathtt{A}$; $turn := 1$
end .

It is easy to see that the body of *Cdb.k* always terminates with $turn.k \neq 3$. So, if key k acts often enough, the program indeed satisfies the fairness constraint

$Fdat$: $\Box\Diamond(turn.k \neq 3)$.

At this point the question is: does this program indeed implement the abstract specification of Section 4? In order to answer this question, we formally define the word "implements" in terms of the observable behaviours of the program and of its specification. A *behaviour* of a program or specification is an infinite execution sequence that satisfies the fairness constraints. In our case, the observable behaviour is the restriction (projection) of the behaviour to the visible variables $inv.k$, $res.k$, and $turn.k$. The question is then whether every observable behaviour of the program is also some observable behaviour of the specification. We prove this with the new technique of eternity extensions and backward invariants presented in Section 2.

6. Verification of the implementation

In order to verify that the code of Section 5 implements the specification of Section 4, we add auxiliary variables to it, both history variables and eternity variables, in a layered manner. The aim is to arrive at a specification that allows an easy projection to the abstract program *STEP*. We do this by recognizing or inserting components of *Env* and *Adb* in the concrete program.

We shall thus again encounter the shared ghost variables `gnr` and `etMem`, and the private ghost variables *evf*, *start*, *val*, *nr*, *sysAb*, with the same types and initializations as before. We introduce some more ghost variables to guide the choice of the eternity variables.

6.1. Eternity approximated by history

The specification of Section 4.4 requires, for every behaviour of the implementation, some value for `etMem` that satisfies the equalities in *SpE.k* whenever key k successfully terminates. The implementation of Section 5 does not directly provide a suitable value for `etMem`. Since `etMem` is an infinite array, we need not determine `etMem` all at once, but can determine its elements one by one. So, in order to approximate array `etMem`, we introduce the shared history variable

$$\texttt{hiMem}: \textbf{array}\ \mathit{Nat}\ \textbf{of}\ \mathit{DbVal}\ ,$$

with $\texttt{hiMem}[0] = \texttt{db0}$ initially. For every successful transaction, we let the resulting database state be recorded instantaneously in array `hiMem` at the moment procedure *endTrans* is called. The prefixes `hi` and `et` refer to `history` and `eternity`. The idea of "approximation" will be formalized in Section 6.2 below.

A second problem with the specification is that the invocations `B` are answered in *SpI.k* by a nondeterministic choice of *val* or *start*, which equals $\perp$ if and only if the starting transaction will eventually fail. This seems to ask for a prophecy variable. Since our formalism does not provide prophecy variables, we introduce, for every key k, a private eternity variable

$$\mathit{etSno}: \textbf{array}\ \mathit{Nat}\ \textbf{of}\ \mathit{Nat},$$

such that $\mathit{etSno}[i] = n$ means that the ith transaction of the key is the nth successful transaction of the system; $\mathit{etSno}[i] = 0$ means that the ith transaction of the key is not successful. In order to approximate *etSno*, we give each key the private history variables

$$\mathit{hiSno}: \textbf{array}\ \mathit{Nat}\ \textbf{of}\ \mathit{Nat};$$
$$\mathit{cnt}: \mathit{Nat} := 0\ ;$$

where *cnt* counts the transactions the key has been involved in.

Procedure *endTrans* is extended to:

procedure *endTrans* () =
  Commit ;
  while *tlist* $\neq \emptyset$ **do**
    ⟨ *remove some o from tlist* ; `db`[*o*] := *pridb* [*o*] ⟩ **od** ;
  releaseAll() ; *res* := `E` ; *turn* := 1
end .

Here, *tlist* is a new private variable to hold the objects o for which `db`[o] has yet to be updated. *Commit* is a big atomic command that only modifies auxiliary variables. With respect to the implementation, it just adds one stuttering. It determines new values of

hiMem, *cnt*, and *hiSno*. Since the computation of hiMem is somewhat involved, it uses a private ghost variable *ldb*, which is not used elsewhere. We also choose this point to include the serialization event *SpS.k* that increments gnr and copies the result to *nr*:

Commit :
  ldb := hiMem[gnr];
  for all $o \in ownset$ **do** $ldb[o] := pridb[o]$ **od**;
  hiMem[gnr + 1] := *ldb*;
  gnr := gnr + 1 ; *nr* := gnr;
  cnt := *cnt* + 1 ; *hiSno*[*cnt*] := gnr ;
  tlist := *ownset* .

The abortion of a transaction is recorded in *hiSno* by

procedure *aborting* ( ) =
  cnt := *cnt* + 1 ; *hiSno*[*cnt*] := 0 ;
  releaseAll( ) ; *res* := A ; *turn* := 1
end .

Here, *hiSno*[*cnt*] becomes 0 since the latest transaction failed.

6.2. Behaviour restrictions and prophecies

The intention that the history variable hiMem approximates the eternity variable etMem is formally expressed by postulating the behaviour restriction

Rq0 : $\quad i \leqslant \mathtt{gnr} \quad \Rightarrow \quad \mathtt{hiMem}[i] = \mathtt{etMem}[i]$.

Here, we universally quantify over the free variable i, which ranges over the natural numbers. Similarly, the relationship between *hiSno* and *etSno* is expressed in the behaviour restriction

Rq1 : $\quad i \leqslant cnt.k \quad \Rightarrow \quad hiSno.k[i] = etSno.k[i]$.

Extension with eternity variables is only sound when every behaviour allows at least one value for the eternity variable for which the behaviour restriction is satisfied, see condition (BR) in Section 2.4, and [7,8]. Since gnr and hiMem are modified only in *endTrans*, the elements of hiMem are written only once and the written elements are those with index $\leqslant$gnr. This implies satisfiability for *Rq0*. Apart from the fact that *hiSno.k* and *cnt.k* are private variables of key k, the argument for *Rq1* is similar.

We use etMem and *etSno* to guide the choice of *val* when the database answers invocation B. So, on top of the eternity extension, we now introduce private history variables *val*, *start*, and *mnr*, which are modified in the private command

Prophecy :
  mnr := *etSno*[*cnt* + 1] ;
  val := (*mnr* > 0 ? etMem[*mnr* − 1] : ⊥) ;
  start := *val* .

Here, we use a conditional expression $(\_?\_:\_)$ as in the language C. Note that, formally, *mnr* and *start* are history variables, but that they serve as prophecy variables. For instance, $etSno[cnt+1]$ holds the value of $hiSno[cnt+1]$ that has not yet been written. One could say that, when the prophet can read eternity, his prophecies become history.

6.3. The state machine

In order to formulate and verify invariants, we have to pin down the atomic commands precisely. We therefore number them and indicate the flow of control by **goto** commands. As usual, absence of **goto** means a jump to the next command. Note that every label stands for a single atomic command that may range over several lines. In the invariants, we use a private variable *pc* as the program counter that holds the label of the next atomic command.

The environment has three atomic commands that it executes repeatedly. In principle, unknown keys cannot have private variables. We model them however with $pc = 0$. It is also convenient to initialize the ghost variables of unknown keys by

$$turn = cnt = nr = mnr = 0\ ,$$
$$start = val = \bot\ .$$

An unknown key becomes known and is initialized by executing the command at $pc = 0$. We initialize $tlist := \emptyset$ for convenience in the invariants. The states with $turn = 1$ and 2 are represented as the locations with $pc = 10$ and 15, respectively. The environment's activity is thus represented by

```
 0   res := A ;   ownset := ∅ ;   tlist := ∅ ;
     turn := 1 ;   goto 10 ;
10   choose evf > 0 ;   nr := 0 ;   sysAb := false ;
     inv := B ;   turn := 3 ;   goto 20 ;
15   evf := evf − 1 ;
     choose inv ∈ Inv ∪ {E, A} with (evf ⩽ 0 ⇒ inv ∈ {E, A}) ;
     accessed := accessed ∪ |inv| ;   turn := 3 ;   goto 20 .
```

The database distributes the invocations at 20 and handles invocation `B` at 21.

```
20   if inv = B then goto 21
     elsif inv = A then goto 40
     elsif inv = E then goto 50
     else goto 30 fi ;
21   Prophecy ;  (see Section 6.2)
     res := B ;   turn := 2 ;   goto 15 .
```

In the ordinary invocations of a successful transaction, we treat the specification variable *val* as a copy of the implementation variable *pridb*. The justification of this will be the most difficult part of the verification. In accordance with alternative (C) of *Adb*, the specification variable *sysAb* is set when *Acquire-L* fails. We thus combine the code

of *owns* and *handle* in

Modify

30 $(iob, iv) := inv$; **if** $iob \in ownset$ **then goto** 33 **fi** ;
31 **if** $\texttt{owner}[iob] = \bot$ **then** $\texttt{owner}[iob] := self$
else $sysAb := true$; **goto** 40 **fi** ;
32 $ownset := ownset \cup \{iob\}$;
$pridb[iob] := \texttt{db}[iob]$;
33 *choose* $(res, w) \in ObN(iv, pridb[iob])$;
$pridb[iob] := w$; **if** $val \neq \bot$ **then** $val[iob] := w$ **fi** ;
$turn := 2$; **goto** 15 .

The conditional modification of *val* in 33 is motivated by specification *SpI.k* in 4.4 and the definition of $DbF(i, \bot)$.

We duplicate the calls of *releaseAll* in *aborting* and *endTrans*. For the ease of the invariants, we replace the **for** loops that cannot be regarded as atomic with **while** loops. The **while** commands 41, 51 and 52 below are the atomic commands to execute the body once and go back to the start of the loop if the guard holds, and to go to the next command if the guard is false.

40 $cnt := cnt + 1$; $hiSno[cnt] := 0$;
41 **while** $ownset \neq \emptyset$ **do**
remove some o *from* $ownset$; $\texttt{owner}[o] := \bot$ **od** ;
42 $res := \texttt{A}$; $accessed := \emptyset$; $mnr := 0$;
$start := \bot$; $val := \bot$; $turn := 1$; **goto** 10 .

After incorporation of *releaseAll*, procedure *endTrans* becomes:

50 *Commit* ; (see Section 6.1)
51 **while** $tlist \neq \emptyset$ **do**
remove some o *from* $tlist$; $\texttt{db}[o] := pridb[o]$ **od** ;
52 **while** $ownset \neq \emptyset$ **do**
remove some o *from* $ownset$; $\texttt{owner}[o] := \bot$ **od** ;
53 $res := \texttt{E}$; $accessed := \emptyset$; $mnr := 0$;
$start := \bot$; $val := \bot$; $turn := 1$; **goto** 10 .

In 42 and 53, the set *accessed* is made empty in accordance with specifications *SpA.k* and *SpE.k* in 4.4. We also reset the ghost variables *mnr* to 0 and *start* and *val* to $\bot$. This is allowed by the specifications and convenient for the verification.

The level of atomicity is justified by two observations. Firstly, although the environment and the database component at any key are different processes that share several private variables, they do not interfere since they are controlled by a single program counter *pc*. Secondly, the only non-auxiliary variables shared by all keys are `db` and `owner`, which are accessed only in 31, 32, 41, 51, and 52. The double access of `owner` in 31 is justified by the atomicity postulated for *Acquire-L*. The accesses in 32, 41, 51, and 52, are single and can therefore be regarded as atomic. Alternatively, the atomicity of the accesses of `db` in 32 and 51 can also be justified by proving (with the invariants *Nq1* and *Nq5* in 6.7 below) that the key holds the lock of the object.

At this point, the reader should have verified that the state machine, when projected to the implementation variables db, owner, *inv*, *turn*, *res*, *ownset*, and *pridb*, faithfully represents program of Section 5 and that its commands 0, 10, and 15 represent command *Env* of the environment (see 4.2). In the next subsection, we deal with the question whether it satisfies the specification of 4.4.

We first investigate the state machine's fairness properties. The only backward jumps in this code are those from 21, 33, 42, and 53 to 10 or 15. The loops in 41, 51 and 52 are bounded by the sizes of the finite sets *tlist* and *ownset*. All other commands end with a forward jump or are followed by a next command. It follows that, for every key k with $pc.k \notin \{0, 10, 15\}$, the database process at k needs to do only a bounded number of steps to establish $pc.k \in \{10, 15\}$. This justifies the concrete fairness assumption

$$Fconc0 : \ \Box\Diamond(pc.k \in \{0, 10, 15\}) \ .$$

In a transaction, the environment eventually chooses all invocations equal to E or A because of *evf*. Since these invocations are answered by termination of the transaction, it follows that all transactions eventually terminate, i.e.

$$Fconc1 : \ \Box\Diamond(pc.k \in \{0, 10\}) \ .$$

In the mechanical proof, this argument requires more attention.

6.4. The global proof and derived proof obligations

At this point we have rewritten the program of Section 5 and decorated it with actions on ghost variables in such a way that it can be compared with the specification of Section 4. We now reduce the correctness of the program to seven proof obligations $Dq0, \ldots, Dq6$. The obligations *Dq0* and *Dq1* are easily settled. The obligations $Dq2, \ldots, Dq6$ are treated in Sections 6.5 up to 6.8.

To summarize what we have done so far, let $K0$ be the parallel composition of the environment *Env* with the implementation *Cbd* of Section 5. In Section 6.1, we extended $K0$ with history variables gnr, hiMem, *cnt*, *hiSno* to a specification $K1$, with a history extension $K0 \rightarrowtriangle K1$. By adding eternity variables etMem and *etSno* with behaviour restrictions *Rq0* and *Rq1*, we got a specification $K2$ with an eternity extension $K1 \rightarrowtriangle K2$. Note that, according to the definition in Section 2.4, all states of $K2$ satisfy the behaviour restrictions *Rq0* and *Rq1*. So, these predicates are invariants, even tautologies. We finally extended $K2$ with history variables *start*, *val*, *nr*, *accessed*, *sysAb*, *mnr*, and actor, say to specification $K3$. The composition yields a refinement extension $K0 \rightarrowtriangle K3$.

Now let L be the abstract specification with next-state relation *STEP* of Section 4.4. In order to compare $K3$ with L, we form the forgetful function $fg : states(K3) \rightarrow states(L)$ that removes all variables that do not occur in L, viz. db, owner, hiMem, *pc*, *iob*, *iv*, *pridb*, *ownset*, *tlist*, *evf*, *cnt*, *hiSno*, *etSno*, and *mnr*. Suppose for the moment that *fg* is a refinement mapping from $K3$ to L. It would follow that the composition of the refinement extension $K0 \rightarrowtriangle K3$ with the refinement mapping *fg* would be a simulation $K0 \rightarrowtriangle L$. This composition treats the visible variables *inv.k*, *res.k*, *turn.k*

as the identity relation. Therefore, Theorem 2 would imply that the concrete program $K0$ implements specification L.

Unfortunately, fg is not a refinement mapping, since $K3$ has states that can do steps that do not correspond via fg to steps in L. We claim that these states do not occur in behaviours of $K3$. We can therefore save the argument by constructing an invariant D of $K3$ such that the restriction of fg to D is a refinement mapping from the restricted specification $K3_D$ to L. Since the invariant yields a simulation $K3 \multimap K3_D$, the previous argument suffices to prove that $K0$ implements the abstract specification L. It thus remains to construct an invariant D of $K3$ such that the restriction of fg to D is a refinement mapping from $K3_D$ to L.

The state space $states(L)$ is spanned by the visible variables $inv.k$, $res.k$, and $turn.k$, and the specification variables `gnr`, `etMem`, $accessed.k$, $start.k$, $val.k$, $nr.k$, and $sysAb.k$. Since these variables are initialized in $K3$ and L in the same way, fg maps $start(K3)$ to $start(L)$.

We now verify that fg maps steps of $K3$ to steps of L. It is easy to see that the instructions at 20, 30, 32, 40, 41, 51, 52 correspond to stuttering steps of L. The instructions 0, 10, 15 are executed by the environment. Therefore, it remains to prove that the instructions at 21, 31, 33, 42, 50, 53 are mapped to steps of L. We show that these instructions are mapped to steps of L because of certain invariants $Dq0, \ldots, Dq6$. Most of these invariants are not proved here, but only announced as proof obligations to be dealt with later.

Predicate $STEP$ of Section 4.4 has the constituent predicates SpB, SpI, SpA, SpE, SpS, and SpC. Several of these predicates contain the conjunct $turn = 3$. We therefore observe that our system has the easy invariant

$$Dq0: \quad pc.k > 15 \quad \Rightarrow \quad turn.k = 3 \ .$$

The step at 21 corresponds to alternative $SpB.k$ because of $Dq0$ and the easy invariant

$$Dq1: \quad pc.k = 21 \quad \Rightarrow \quad inv.k = \mathtt{B} \ .$$

Instruction 31 is mapped to a stuttering step if $\mathtt{owner}[iob.k] = \bot$. Otherwise it contains an assignment to $sysAb$, which corresponds to the alternative $SpC.k$ if we have the invariant

$$Dq2: \quad pc.k = 31 \ \wedge \ \mathtt{owner}[iob.k] \neq \bot \quad \Rightarrow \quad conflict(k) \ .$$

We claim that instruction 33 corresponds to alternative $SpI.k$. This gives us the obligation to prove that $(res, val)^+ \in DbF(inv, val)$ holds at instruction 33. If $val = \bot$ then $val^+ = \bot$ and hence $(res, val)^+ \in DbF(inv, val)$ because of the definition of $DbF(i, \bot)$. Since $inv = (iob, iv)$, the definition of DbF in terms of ObN implies that it suffices to prove that, if $val \neq \bot$, instruction 33 satisfies

$$\begin{array}{l} (res^+, val^+[iob]) \in ObN(iv, val[iob]) \\ \wedge \quad (\forall\ p : p \neq iob : val[p] = val^+[p]) \ . \end{array}$$

This would follow from $val[iob] = pridb[iob]$. It therefore suffices to prove the invariant

$$Dq3: \quad pc.k = 33 \;\Rightarrow\; inv.k = (iob.k, iv.k) \\ \wedge\, (val.k = \bot \vee val.k[iob.k] = pridb.k[iob.k]) \;.$$

Instruction 42 satisfies $SpA.k$ (abortion) if we have the invariant

$$Dq4: \quad pc.k = 42 \;\Rightarrow\; nr.k = 0 \;\wedge\; (inv.k = \mathtt{A} \vee sysAb.k) \;.$$

Instruction 50 corresponds to $SpS.k$ (serialization) if we have the invariant

$$Dq5: \quad pc.k = 50 \;\Rightarrow\; nr.k = 0 \;.$$

Instruction 53 corresponds to $SpE.k$ (termination) if we have the invariant

$$Dq6: \quad pc.k = 53 \;\Rightarrow\; inv.k = \mathtt{E} \;\wedge\; nr.k > 0 \\ \wedge\, start.k = \mathtt{etMem}[nr.k - 1] \;\wedge\; val.k = \mathtt{etMem}[nr.k] \;.$$

Let D be the universal quantification over all keys k of the conjunction of $Dq0$ up to $Dq6$. Function fg maps every step of specification $K3$ that starts in a state of D to a step of the abstract specification L. Every behaviour of $K3$ is therefore mapped to an initial execution of L. Every behaviour of $K3$ satisfies property $Fconc0$ and hence $Fdat$ because of $Dq0$, and is therefore mapped to a behaviour of L. This implies that fg is a refinement mapping from the restricted specification $K3_D$ to L.

The easy invariance of $Dq0$ and $Dq1$ was noticed above. In the next sections we prove that $Dq2$, $Dq3$, $Dq4$, $Dq5$, $Dq6$ are indeed invariants. We first apply standard techniques that yield forward invariants and prove $Dq2$, $Dq4$, and $Dq5$. The remaining cases $Dq3$ and $Dq6$ need the behaviour restrictions and backward invariants, i.e., applications of Lemmas 0 and 1 of Section 2.2.

As sketched in Section 7 below, the claim that the above proof obligations $Dq0$ up to $Dq6$ together imply that the state machine of Section 6.3 satisfies specification $STEP$ of Section 4.4 is mechanically proved in Part 3 of the mechanical proof `sdi.events` [9].

6.5. The forward invariants

In this section we settle the proof obligations $Dq2$, $Dq4$, $Dq5$, and prove some invariants needed for $Dq3$ and $Dq6$. All invariants mentioned in this section are forward invariants, proved by standard techniques. For simplicity in the invariants, we use the convention that $ownset.k$ and $tlist.k$ are empty for unknown keys. The results of this subsection are mechanically proved in Part 4 of `sdi.events` [9].

Predicate $Dq2$ asserts the existence of a conflict when some key, say q, is at 31 and $r = \mathtt{owner}.(iob.q)$ is a genuine key ($r \neq \bot$). We prove this conflict by showing that $iob.q$ is a common element of $accessed.q$ and $accessed.r$ and that $q \neq r$. The first assertion follows from the invariants

$$Jq0: \quad pc.k \in \{31, 32, 33\} \;\Rightarrow\; iob.k \in accessed[k] \;, \\ Jq1: \quad \mathtt{owner}[o] = k \neq \bot \;\Rightarrow\; o \in accessed[k].$$

Note that in such invariants we universally quantify over all free variables (here k and o). The inequality $q \neq r$ follows from the easy invariants

$$Jq2: \quad pc.k \in \{31,32\} \;\Rightarrow\; iob.k \notin ownset.k \;,$$
$$Jq3: \quad \texttt{owner}[o] = k \neq \bot \;\wedge\; o \notin ownset.k \;\Rightarrow\; pc.k = 32 \;\wedge\; o = iob.k \;.$$

Indeed, *Dq2* follows from *Jq0*, *Jq1*, *Jq2*, and *Jq3*. It therefore remains to prove that these predicates are invariants. They hold initially, since then $pc.k = 0$ and $\texttt{owner}[o] = \bot$. Predicate *Jq0* is preserved at 30 because of the easy invariant

$$Jq4: \quad pc.k \in \{20,30\} \;\wedge\; inv = (o,i) \;\Rightarrow\; o \in accessed[k] \;.$$

Jq1 is preserved at 31 because of *Jq0*. Its preservation at 42 and 53 follows from *Jq3* together with the easy invariant that *ownset* is empty outside and at the end of transactions:

$$Jq5: \quad pc.k \in \{0,10,42,53\} \;\Rightarrow\; ownset.k = \emptyset \;.$$

Preservation of *Jq2* and *Jq3* is straightforward. This concludes the proof of *Dq2*.

We turn to the proofs of *Dq4* and *Dq5*. With respect to *nr*, it is easy to see the invariance of

$$Kq0: \quad pc.k \notin \{0,10,51,52,53\} \;\Rightarrow\; nr.k = 0 \;.$$

Since a key can only arrive at 40 from 20 or 31, we also have the easy invariant

$$Kq1: \quad pc.k \in \{40,41,42\} \;\Rightarrow\; inv.k = \texttt{A} \vee sysAb.k \;.$$

Clearly, *Dq4* follows from *Kq0* and *Kq1*, while *Dq5* follows from *Kq0*.

It remains to consider *Dq3* and *Dq6*. For *Dq3*, it is easy to prove the invariants

$$Kq2: \quad pc.k \in \{30,31,32,33\} \;\Rightarrow\; inv.k \in Inv \;,$$
$$Kq3: \quad pc.k \in \{31,32,33\} \;\Rightarrow\; inv.k = (iob.k, iv.k) \;.$$

Preservation of *Kq3* at 30 follows from *Kq2*. We postpone the treatment of the second conjunct of *Dq3*, since it involves the variable *val*.

The easy part of *Dq6* consists of the two easy invariants

$$Kq4: \quad pc.k \in \{50,51,52,53\} \;\Rightarrow\; inv.k = \texttt{E} \;,$$
$$Kq5: \quad pc.k \in \{51,52,53\} \;\Rightarrow\; nr.k = hiSno.k[cnt.k] > 0 \;.$$

It remains to analyse the values of *start* and *val* at 33 and 53 for *Dq3* and *Dq6*.

6.6. Reduction of the proof obligations

In this section, we settle the remaining proof obligations under assumption of one postulated invariant *Yq0*. The assertions in this subsection have been formally proved in Part 5 of `sdi.events` [9].

Since *start* is modified less often than *val*, we begin with the analysis of *start*. Variable *start* gets its value at 21 by means of *mnr*. We therefore first prove

the straightforward invariants

$$Lq0: \quad start.k = (mnr.k = 0 \,?\, \bot : \texttt{etMem}[mnr.k - 1]) \ ,$$
$$Lq1: \quad pc.k \in \{15, 30, 31, 32, 33, 40, 50\} \vee (pc.k = 20 \ \wedge \ inv.k \neq \texttt{B}) \Rightarrow \ mnr.k = etSno.k[cnt.k + 1] \ ,$$
$$Lq2: \quad pc.k \in \{41, 42, 51, 52, 53\} \ \Rightarrow \ mnr.k = etSno.k[cnt.k] \ .$$

Preservation of *Lq2* at 40 and 50 follows from *Lq1*.

At this point, we use behaviour restriction *Rq1* which says that $etSno.k[i] = hiSno.k[i]$ for all $i \leqslant cnt.k$. Together with the invariants *Lq2* and *Kq5*, this implies the derived invariant

$$Aq0: \quad pc.k \in \{51, 52, 53\} \ \Rightarrow \ mnr.k = nr.k \ .$$

Clearly, *Lq0*, *Aq0*, and *Kq5* together imply

$$Aq1: \quad pc.k \in \{51, 52, 53\} \ \Rightarrow \ start.k = \texttt{etMem}[nr.k - 1] \ .$$

This is the assertion about *start* in *Dq6*. Below, we also need that *start* differs from $\bot$ at these locations. Since behaviour restriction *Rq0* gives $\texttt{etMem}[i] = \texttt{hiMem}[i]$ for $i \leqslant \texttt{gnr}$, we note the easy invariants

$$Lq3: \quad nr.k \leqslant \texttt{gnr} \ ,$$
$$Lq4: \quad i \leqslant \texttt{gnr} \ \Rightarrow \ \texttt{hiMem}[i] \neq \bot \ .$$

Using *Aq1* together with *Rq0*, *Lq3*, and *Lq4*, we thus obtain

$$Aq2: \quad pc.k \in \{51, 52, 53\} \ \Rightarrow \ start.k \neq \bot \ .$$

The key to the analysis of *val* for *Dq3* and *Dq6* is the invariant

$$Yq0: \quad o \in ownset.k \ \wedge \ val.k \neq \bot \ \Rightarrow \ val.k[o] = pridb.k[o] \ .$$

The proof of this invariant will require backward invariants and is therefore postponed to the next subsection. It is clear that *Dq3* follows from *Yq0*, *Kq3*, and the easy invariant

$$Lq5: \quad pc.k = 33 \ \Rightarrow \ iob.k \in ownset.k \ .$$

On the other hand, *Dq6* is easily seen to follow from *Kq4*, *Kq5*, *Aq1*, and the postulate

$$Yq1: \quad pc.k \in \{52, 53\} \ \Rightarrow \ val.k = \texttt{etMem}[nr.k] \ .$$

Invariance of *Yq1* is threatened only at 51. It is preserved at 51 because of the derived invariant

$$Aq3: \quad pc.k = 51 \ \Rightarrow \ val.k = \texttt{etMem}[nr.k] \ .$$

Predicate *Aq3* is proved by considering the values of the separate objects in the databases. We first claim the invariants

$$Mq0: \quad val.k = \bot \ \equiv \ start.k = \bot \ ,$$
$$Mq1: \quad start.k \neq \bot \ \wedge \ o \notin ownset.k \ \wedge \ pc.k \notin \{41, 42, 52, 53\} \Rightarrow \ val.k[o] = \texttt{etMem}[mnr.k - 1][o] \ .$$

Preservation of *Mq0* is easy. Preservation of *Mq1* at 33 follows from *Lq5*.

We now observe that *Mq0*, *Mq1*, *Yq0*, and *Aq2*, together imply

$$pc.k = 51 \;\Rightarrow\; val.k[o] = (o \in ownset.k \,?\, pridb.k[o] : etMem[mnr.k - 1][o]) \;.$$

On the other hand, we have the complicated but straightforward invariant

$$Mq2: \quad pc.k = 51 \;\Rightarrow\; \texttt{hiMem}[nr.k][o] = (o \in ownset.k \,?\, pridb.k[o] : \texttt{hiMem}[nr.k - 1][o]) \;.$$

Indeed, *Mq2* precisely expresses the assignment to `hiMem` in 50.

Now *Aq3* follows by comparison with *Mq2*, using *Aq0*, *Rq0*, *Kq5*, and *Lq3*. At this point, we use that the databases *val.k* and `etMem`[*nr.k*] are functions on objects which are equal if (and only if) they yield the same values for every object argument. This concludes the proof of *Aq3* and thus of *Yq1* and *Dq6*.

6.7. Preparation of the proof of Yq0

In preparation of the proof of *Yq0*, we note that *pridb* gets its values from the shared variable `db`. So, we need to relate `db` with the ghost variables. The assertions in this subsection have been formally proved in Part 6 of `sdi.events` [9]. We claim the invariant

$$Nq0: \quad o \notin tlist.(\texttt{owner}[o]) \;\Rightarrow\; \texttt{db}[o] = \texttt{hiMem}[\texttt{gnr}][o] \;.$$

Here, we introduce the convention that *tlist.k* and *ownset.k* are always empty for the non-existing key $\perp$. So, if object o has no owner, *Nq0* asserts $\texttt{db}[o] = \texttt{hiMem}[\texttt{gnr}][o]$. Predicate *Nq0* holds initially, since then *tlist.k* is empty and $\texttt{db} = \texttt{db0} = \texttt{hiMem}[0]$ and $\texttt{gnr} = 0$. Predicate *Nq0* is preserved in 50, 41, and 52 because of the invariants

$$Nq1: \quad o \in ownset.k \;\Rightarrow\; \texttt{owner}[o] = k \;,$$
$$Nq2: \quad tlist.k = \emptyset \;\vee\; pc.k = 51 \;.$$

It is preserved at 51 because of the invariants

$$Nq3: \quad tlist.k \subseteq ownset.k \;,$$
$$Nq4: \quad pc.k = 51 \;\wedge\; o \in ownset.k \;\Rightarrow\; pridb.k[o] = \texttt{hiMem}[\texttt{gnr}][o] \;.$$

Predicate *Nq1* justifies the **assert** in *Release-L* of Section 5. It is an invariant since it is preserved at 32 because of

$$Nq5: \quad pc.k \in \{32, 33\} \;\Rightarrow\; \texttt{owner}[iob.k] = k \;.$$

Preservation of *Nq2* is easy. Preservation of *Nq3* at 41 and 52 follows from *Nq2*. Predicate *Nq4* is preserved at 50 because of *Nq1*. Predicate *Nq5* is preserved at 30, 41, and 52 because of *Nq1*.

In the proof of *Yq0*, we also need two other relatively easy invariants. Firstly, as a variation and consequence of *Jq5*, we have

$$Pq0: \quad pc.k = 21 \vee (pc.k = 20 \;\wedge\; inv.k = \texttt{B}) \;\Rightarrow\; ownset.k = \emptyset \;.$$

As a variation of *Kq5* we have the invariant

$$Pq1: \quad pc.k \in \{41,42\} \;\Rightarrow\; hiSno.k[cnt.k] = 0 \;.$$

If $pc.k \in \{41,42\}$, then *Lq2*, *Rq1*, and *Pq1* imply that $mnr.k = etSno.k[cnt.k] = hiSno.k[cnt.k] = 0$. By *Lq0*, this gives us $start.k = \bot$, thus proving the derived invariant

$$Aq4: \quad pc.k \in \{41,42\} \;\Rightarrow\; start.k = \bot \;.$$

The mechanical proof for the backward invariants below requires progress. This is based on the invariant

$$Pq2: \quad pc.k = 15 \;\Rightarrow\; evf.k > 0 \;.$$

Preservation of *Pq2* at 21 and 33 follows from *Dq1*, *Kq2* and the easy invariant

$$Pq3: \quad pc.k \notin \{0,10\} \;\wedge\; evf.k = 0 \;\Rightarrow\; inv.k \in \{\mathtt{E},\mathtt{A}\} \;.$$

To summarize, using the star * as a wild card, all predicates *Dq0*, *Dq1*, *Dq2*, *Dq4*, *Dq5*, *Jq**, *Kq**, *Lq**, *Mq**, *Nq**, *Pq** are forward invariants, independently of the behaviour restrictions *Rq0* and *Rq1*. The behaviour restrictions and the unproved invariant *Yq0* have only been used in the proofs of the derived invariants *Aq** and in the proof of invariance of *Yq1*.

6.8. Backward invariants

We turn to the proof of *Yq0*. We leave the comfortable realm of forward invariants. So, all invariants introduced in this section are not forward invariants. The results of this subsection are mechanically proved in Parts 7 and 9 of `sdi.events` [9].

The invariance of *Yq0* is proved by means of Lemma 0 of Section 2.2, but the auxiliary invariant needed will be proved with Lemma 1. Predicate *Yq0* holds initially since then *ownset* is empty. It is preserved at 21 because of *Pq0*. Preservation of *Yq0* at 32 follows from *Mq0* and the new postulate

$$Aq5: \quad pc.k = 32 \;\wedge\; start.k \neq \bot \;\Rightarrow\; val.k[iob.k] = \mathtt{db}[iob.k] \;.$$

The left-hand side of the consequent of *Aq5* equals $\mathtt{etMem}[mnr.k - 1][iob.k]$ because of *Jq2* and *Mq1*. The right-hand side equals $\mathtt{hiMem}[\mathtt{gnr}][iob.k]$ because of *Nq0*, *Nq2*, and *Nq5*. Therefore, *Aq5* is equivalent to the predicate

$$Qz0: \quad pc.k = 32 \;\wedge\; start.k \neq \bot \;\Rightarrow\; \mathtt{etMem}[mnr.k - 1][iob.k] = \mathtt{hiMem}[\mathtt{gnr}][iob.k] \;.$$

This predicate is not a forward invariant, but it is an invariant. This is proved by means of Lemma 1. Indeed, it is implied by the attractor of *Fconc0*, since that implies that always eventually $pc.k \neq 32$. Backward stability is verified by means of the second backward invariant

$$Qz1: \quad start.k \neq \bot \;\wedge\; o \in ownset.k \;\wedge\; pc.k \notin \{51,52\} \;\Rightarrow\; \mathtt{etMem}[mnr.k - 1][o] = \mathtt{hiMem}[\mathtt{gnr}][o]$$

and the implication

(B0) $\quad Qz0^+ \ \wedge \ Qz1^+ \ \wedge \ Nq1 \ \wedge \ Nq5 \ \Rightarrow \ Qz0$.

Implication (B0) is proved as follows. Assume that $Qz0$ does not hold and that key p does a step that establishes $Qz0$, while $Qz1$ also holds in the postcondition. First, assume $p \neq k$. Then $pc.k = 32$ and p must execute 50 to modify $\texttt{hiMem}[\texttt{gnr}][iob.k]$. This implies $iob.k \in ownset.p$. Then $Nq1$ and $Nq5$ together imply $p = \texttt{owner}[iob.k] = k$. It remains to assume $p = k$. Then the step modifies $pc.k$ and establishes $Qz0$. This implies that k executes 32 in a state with $start.k \neq \perp$ while the consequent of $Qz0$ is false. Since this step adds $iob.k$ to $ownset.k$, we see that $Qz1$ is false in the postcondition, contradicting the assumption. This proves (B0).

To prove that $Qz1$ is a backward invariant, we first note that it is implied by the attractor of $Fconc1$ because of $Jq5$: since $pc.k \in \{0, 10\}$ holds infinitely often, $ownset$ is infinitely often empty, and then $Qz1$ holds, for given k and o. Backward stability of $Qz1$ is expressed by the implication

(B1) $\quad Qz1^+ \ \wedge \ Rq1^+ \ \wedge \ Rq0 \ \wedge \ Jq5 \ \wedge \ Lq1 \ \wedge \ Nq1 \ \wedge \ Pq0 \ \wedge \ Aq4$
$\Rightarrow \ Qz1$,

which is proved as follows. Given k and o, suppose $Qz1$ is false and is established by a step of key p. First assume $p \neq k$. Then the antecedent of $Qz1$ holds in the precondition and in the postcondition, and key p executes 50 to modify $\texttt{hiMem}[\texttt{gnr}][o]$. It follows that object o is in $ownset.k$ and in $ownset.p$. Now $Nq1$ implies $p = k$. Therefore, assume $p = k$. Since $Qz1$ is false in the precondition, the antecedent of $Qz1$ then holds. By $Jq5$ and $Pq0$, this implies that $pc.k \notin \{0, 10, 21, 42, 53\}$. Also, $pc.k \notin \{51, 52\}$, and $pc.k \neq 41$ by $Aq4$. It remains to consider $pc.k = 50$. The precondition has $mnr.k = etSno.k[cnt.k + 1]$ by $Lq1$. In terms of the precondition, command 50 sets $hiSno.k[cnt.k + 1]$ to $\texttt{gnr} + 1$. Using that $Rq1$ holds in the postcondition, we see that the precondition satisfies $mnr.k - 1 = \texttt{gnr}$. Since $\texttt{etMem}[\texttt{gnr}] = \texttt{hiMem}[\texttt{gnr}]$ by $Rq0$, this shows that $Qz1$ holds in the precondition, a contradiction that proves (B1).

So, by Lemma 1, the predicates $Qz0$ and $Qz1$ are both invariants. Consequently, $Aq5$ and $Yq0$ are also invariants. This concludes the correctness proof of the algorithm.

7. The mechanical proof

In order to illustrate the feasibility of the approach and to verify the validity of the results, I undertook the construction of a mechanical proof with the theorem prover NQTHM 1992 of [3]. Actually, I first used it as the standard tool to verify invariants, and I stopped at the emergence of eternity variables since I mistakenly thought that eternity variables would require a higher-order logic not available in NQTHM. At MPC'02, Ernie Cohen convinced me, however, that it could and should be done with the same prover.

The resulting mechanical proof [9] serves as a witness for the soundness and feasibility of our approach. It also sheds light on some of the formal details of the argument.

We use the set-up for concurrency verification that we described in [6]. This set-up had to be extended at several points. Since the values of the eternity variables depend on the specific behaviour, we need the behaviour to construct the values of the eternity variables and we need these values to construct the behaviour. For humans, this raises the danger of circular reasoning, but one cannot fool a sound theorem prover like NQTHM.

7.1. A behaviour and its extension

To avoid circular reasoning, we distinguish three levels for the variables in the system. The level *Le0* of the concrete implementation has the shared variables `db` and `owner`, and the private variables *inv*, *res*, *turn*, *pc*, *evf*, *pridb*, *iob*, *iv*, *ownset*, *tlist*.

The second level *Le1* is the history extension with the shared variables `gnr` and `hiMem` and the private variables *cnt* and *hiSno*, which serve to approximate the eternity variables `etMem` and *etSno*. In the mechanical theory, we regard these eternity variables not as part of the state space, but as constants or rather, since they are infinite arrays, as externally given functions.

The third level *Le2* is again a history extension, now with the remaining ghost variables *accessed*, *sysAb*, *nr*, *mnr*, *start*, *val*, and `actor`.

The first extension of our set-up for the theorem prover was to include the possibility to project a program and its state space to a subset of the spanning variables when the modifications of retained variables only depend on retained variables, and then to prove that the program steps project correspondingly. It is easy to prove that the three levels are correct extensions: the modifications of variables in level *Le.i* only depend on variables in *Le.i*.

We postulate some scheduling function $round : \mathbb{N} \to Keys$ with the property that every occurring key is scheduled infinitely often. This means that, for every $n \in \mathbb{N}$, the set of indices i with $round(i) = round(n)$ is infinite. We then construct a corresponding behaviour xs of the system of level *Le1* such that xs_0 is the initial state of *Le1* and that state xs_{n+1} is obtained from xs_n by a step of key $k = round(n+1)$. For the ease of the proof of progress, the modelling ensures that this step is nonstuttering unless $pc.k = 10$.

In order to define the eternity variables `etMem` and *etSno*, we prove, for all numbers i, m, n and keys k,

$$i \leqslant \mathtt{gnr}(xs_m) \wedge i \leqslant \mathtt{gnr}(xs_n) \Rightarrow \mathtt{hiMem}[i](xs_m) = \mathtt{hiMem}[i](xs_n) \ ,$$
$$i \leqslant cnt.k(xs_m) \wedge i \leqslant cnt.k(xs_n) \Rightarrow hiSno.k[i](xs_m) = hiSno.k[i](xs_n) \ .$$

Here, we regard these program variables of level *Le1* as state functions, which are applied to the states xs_m and xs_n. These implications imply that we can define functions `etMem` and *etSno.k* on the natural numbers such that, for all numbers i, m and keys k,

$$Rq0(xs_m) : \quad i \leqslant \mathtt{gnr}(xs_m) \quad \Rightarrow \quad \mathtt{hiMem}[i](xs_m) = \mathtt{etMem}(i) \ ,$$
$$Rq1(xs_m) : \quad i \leqslant cnt.k(xs_m) \quad \Rightarrow \quad hiSno.k[i](xs_m) = etSno.k(i) \ .$$

In other words, for given behaviour xs, there exist values for `etMem` and *etSno* that satisfy the behaviour restrictions *Rq0* and *Rq1*. Actually, this step requires application

of a form of the axiom of choice, the soundness of which we cannot prove with the theorem prover NQTHM.

The functions `etMem` and *etSno* obtained in this way are now used in the system of level *Le2*. Let *ys* be the behaviour of this system obtained by the same scheduling and the same nondeterministic choices as *xs*. It is easy to see that, for every n, state xs_n is the projection of ys_n to the state space of *Le1*.

7.2. The relational specification satisfied

As in our previous mechanical proofs, the state machine of Section 6.3 is treated as an essentially deterministic automaton, but for the occurrence of a hidden variable `oracle` that is consulted and modified whenever a nondeterministic choice has to be made, e.g. see [6, 1.5]. This is more convenient than a relational representation since with this set-up the verification of invariants is a matter of rewriting. It also makes it relatively easy to verify that the algorithm used in the prover correctly represents the algorithm discussed here.

In order to mechanize the argument given in Section 6.4, we develop a method to express the specification of Section 4.4 syntactically, in such a way that it can be interpreted by the prover. For this purpose, we define an NQTHM function `eval2` that can interpret relations between the state space and the new state space and that can interpret function symbols `unchanged` for Ξ, and `or` for $\vee$, and `and` for $\wedge$, of arbitrary arity. This function is specialized to a function `interpret` for the local view of a single process (key).

The specification *STEP* of Section 4.4 is represented by the definition of a syntactic constant `sdi-spec` with constituents for the six predicates *SpB* up to *SpC*. For example, predicate *SpB* is represented by

```
(defn sp-begin ()
   '(and (equal turn 3)
         (equal (new turn) 2)
         (unchanged gnr accessed nr sysAb)
         (equal inv 'begin)
         (equal (new res) 'begin)
         (equal (new start) (new val)) ) )
```

The new-state operator, used in $turn^+$, res^+, etc., is represented by the function symbol `new`. The symbol `B` is represented by `'begin`. The semantic function that relates the old state `x` with the new state `y` as seen from process `q` is defined by

```
(sdinext q x y) = (interpret (sdi-shared) (sdi-spec) q x y)
```

where `sdi-shared` is the list of shared variables `db`, `owner`, `gnr`, `hiMem`, and `actor`.

Part 3 of the mechanical proof `sdi.events` in [9] proves that the proof obligations *Dq0* up to *Dq6* are enough to prove that the implementation of Section 6.3 satisfies step relation `sdinext`.

7.3. Analysis of steps

The standard methods of concurrency verification, see e.g. [5,6,13,15,17], easily yield that all states ys_n of the behaviour *ys* satisfy the forward invariants of Section 6.5, and

in particular the proof obligations *Dq0*, *Dq1*, *Dq2*, *Dq4*, and *Dq5*. Actually, we had to use some new techniques to allow for unboundedly many keys, but these were not very surprising and, presumably, comparable to those used in [16].

Standard techniques were also sufficient to prove the two backward stability assertions (B0) and (B1) of Section 6.8. Apart from the implicit dependence via the eternity variables, all these results are independent of the particular behaviour.

7.4. Progress formalized with expanding functions

The next step is the application of Lemma 2 of Section 2.2. With a higher-order prover, e.g., as PVS, it would be preferable to prove Lemma 2 first and then apply it. With NQTHM, however, we prefer to prove the particular application directly, since the existential quantification hidden in the condition $Beh(K) \subseteq \Box\Diamond[\![A]\!]$ must be expressed constructively. Our constructive formalization is based on expanding functions, defined as follows.

A function $f : \mathbb{N} \to \mathbb{N}$ is called *expanding* iff $n \leqslant f(n)$ for all n. It is called *strictly expanding* iff $n < f(n)$ for all n. Most of our progress properties are proved with

Theorem. *Let functions* P, $Q : \mathbb{N} \to \mathbb{B}$ *and* g, $vf : \mathbb{N} \to \mathbb{N}$ *be given such that* g *is expanding and*

(a) $P(n) \;\Rightarrow\; P(n+1) \vee Q(n+1)$,
(b) $P(n) \;\Rightarrow\; vf(n) \geqslant vf(n+1)$,
(c) $P(n) \wedge P(g(n)) \;\Rightarrow\; vf(g(n)) > vf(g(n+1))$.

Then there is an expanding function G *with* $P(n) \Rightarrow Q(G(n))$, *which can be constructed explicitly*.

This theorem is mathematically fairly obvious and can also be readily proved with NQTHM.

The requirement that, for every $n \in \mathbb{N}$, the set of indices i with $round(i) = round(n)$ is infinite, is formalized by postulating the existence of a strictly expanding function $h : \mathbb{N} \to \mathbb{N}$ with $round(h(n)) = round(n)$ for all n.

For any key k, the fairness condition *Fconc0*: $\Box\Diamond(pc.k \in \{0, 10, 15\})$ is formalized by means of an expanding function H with the property $pc.k(ys_{H(n)}) \in \{0, 10, 15\}$ for all numbers n. This is proved by a direct application of the above theorem with $vf(n) = vfk(ys_n)$ where the state function vfk is given by

$$\begin{aligned} vfk \;=\; & (\; pc.k \in \{20, 50\} \;?\; 2 \cdot (\#ownset.k) + 60 - pc.k \\ & :\; pc.k = 51 \;?\; \#ownset.k + \#tlist.k + 60 - pc.k \\ & :\; pc.k \in \{30, 31, 40, 41, 42, 52, 52\} \;?\; \#ownset.k + 60 - pc.k \\ & :\; pc.k \in \{32, 33\} \;?\; 60 - pc.k \\ & :\; pc.k \in \{0, 10, 15\} \;?\; 0 \;)\;. \end{aligned}$$

It is clear that vfk remains constant when some key $\neq k$ takes a step and that it decreases when $pc.k \notin \{10, 15\}$ and key k itself takes a step.

For any key k, the fairness condition *Fconc1*: $\Box\Diamond(pc.k \in \{0, 10\})$ is formalized and sharpened by constructing an expanding function K with the property

$$pc.k(ys_n) \neq 0 \quad \Rightarrow \quad pc.k(ys_{K(n)}) = 10 \ .$$

The mechanical proof of this result is based on the above theorem, the previous result, and the ghost variable *evf* that is decremented by instruction 15. The invariant *Pq2* implies that it can be used to construct a variant function.

In order to prove validity of backward invariants, we prove the following constructive version of Lemma 2.

Theorem. *Consider functions* $Q, R : \mathbb{N} \to \mathbb{B}$ *such that* $Q(n+1) \Rightarrow Q(n)$ *and* $R(n) \Rightarrow Q(n)$ *for all* n. *Let* g *be an expanding function such that* $\neg R(n) \Rightarrow Q(g(n))$ *for all* n. *Then* $Q(n)$ *holds for all* $n \in \mathbb{N}$.

This theorem is used in conjunction with (B0) and (B1) to prove that all states ys_n satisfy the predicates *Qz0* and *Qz1*. Using standard arguments we finally obtain that all states ys_n satisfy the other invariants mentioned in Section 6.8 and the proof obligations *Dq3* and *Dq6*. This completes the proof that behaviour *ys* satisfies the specification of Section 4.4.

7.5. Overview of the proof

The proof consists of two NQTHM event files `newprelude` and `sdi`. Such event files are written by the human verifier and verified by NQTHM. They contain all hints the prover needs for the verification. So, a human reader with access to the prover can easily inspect all details of the proof. The file `newprelude` is the prelude for shared-variable concurrency. It has about 800 lines, mainly devoted to extension with history variables. The file `sdi` has about 4960 lines.

The first 800 lines of `sdi` are devoted to the algorithm and its two behaviours as discussed in Section 7.1. Roughly 600 lines are needed for the semantic lemmas that describe how each variable is modified in every step. The syntactic form of the algorithm is easy to compare with the description in this paper. The semantic lemmas make the verifications of invariants much faster and much more convenient. Another 500 lines are devoted to the specification as described in Section 7.2. This part culminates in the seven proof obligations *Dq**.

The main effort is in the standard analysis described in Section 7.3. It takes 400 lines to construct and prove the forward invariants of Section 6.5, Section 6.6 requires around 350 lines. The treatment of Section 6.7 together with the construction and initialization of one global strong invariant requires 870 lines. The proof of *Yq0* and the implications (B0) and (B1) of Section 6.8 require roughly 500 lines. We finally need another 700 lines to prove progress and 200 lines for the validity of the backward invariants as discussed in Section 7.4. This part concludes with the final correctness assertion that the nth step of behaviour `ys` as induced by the acting key `(round (add1 n))` is

conform the specification:

```
(sdinext (round (add1 n)) (ys n) (ys (add1 n)))
```

Indeed, we count the states from 0 and the steps from 1.

NQTHM's verification of the complete events file takes less than nine minutes on a Pentium 4. After that, minor modifications of the proof can be verified or falsified much faster.

8. Conclusions

For the verification of refinement in concurrency, eternity variables form an important and viable mechanism. In our case, the main burden is still done with history variables and (standard) forward invariants, but critical parts are verified with eternity variables. Here safety can rely on progress arguments via backward invariants.

We needed eternity variables in the specification of serializability since serializability is expressed in terms of complete behaviours. It is therefore not surprising that we also needed eternity variables in the proof. A second array of eternity variables *etSno* was needed, however, to connect private data with the shared database.

The feasibility of the approach and the validity of the results are witnessed by the mechanical verification [9] with the first-order theorem prover NQTHM.

Acknowledgements

I am grateful to Eerke Boiten for the opportunity to talk about these ideas in Canterbury, to Ernie Cohen for convincing me that mechanical verification with eternity variables needs no higher order logic, but can be done with a first order theorem prover, and to three anonymous referees for their constructive criticisms.

References

[1] M. Abadi, L. Lamport, The existence of refinement mappings, Theoret. Comput. Sci. 82 (1991) 253–284.
[2] M. Abadi, G.D. Plotkin, A logical view of composition, SRC Research Report 86, Digital Systems Research Center, 1992.
[3] R.S. Boyer, J.S. Moore, A Computational Logic Handbook, 2nd Ed., Academic Press, New York, 1997.
[4] M. Broy, Algebraic and functional specification of an interactive serializable database interface, Distrib. Comput. 6 (1992) 5–18.
[5] W.H. Hesselink, The verified incremental design of a distributed spanning tree algorithm: extended abstract, Formal Aspects Comput. 11 (1999) 45–55.
[6] W.H. Hesselink, An assertional criterion for atomicity, Acta Inform. 38 (2002) 343–366.
[7] W.H. Hesselink, Eternity variables to simulate specifications, in: E.A. Boiten, B. Möller (Eds.), Mathematics of Program Construction, Proc. MPC, Lecture Notes in Computer Science, vol. 2386, Springer, Berlin, 2002, pp. 117–130.
[8] W.H. Hesselink, Eternity variables to prove simulation of specifications, ACM Trans. Comput. Logic, to appear. http://www.acm.org/tocl/accepted.html
[9] W.H. Hesselink, www.cs.rug.nl/∼wim/mechver/eternity contains the two NQTHM events files newprelude and sdi.

[10] R. Kurki-Suonio, Operational specification with joint actions, Distrib. Comput. 6 (1992) 19–37.
[11] S.S. Lam, A.U. Shankar, Specifying modules to satisfy interfaces: a state transition system approach, Distrib. Comput. 6 (1992) 39–63.
[12] L. Lamport, Critique of the Lake Arrowhead three, Distrib. Comput. 6 (1992) 65–71.
[13] L. Lamport, The temporal logic of actions, ACM Trans. Programming Lang. Systems 16 (1994) 872–923.
[14] N.A. Lynch, Distributed Algorithms, Morgan Kaufman, San Francisco, 1996.
[15] Z. Manna, A. Pnueli, Temporal Verification of Reactive Systems: Safety, Springer, Berlin, 1995.
[16] J.S. Moore, G. Porter, The apprentice challenge, ACM Trans. Programming Lang. Systems 24 (2002) 193–216.
[17] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Inform. 6 (1976) 319–340.
[18] F.B. Schneider, Introduction, Distrib. Comput. 6 (1992) 1–3.