Refinement Verification of the Lazy Caching Algorithm

Wim H. Hesselink, 21st February 2006
Dept. of Mathematics and Computing Science, Rijksuniversiteit Groningen
P.O.Box 800, 9700 AV Groningen, The Netherlands
Email: wim@cs.rug.nl, Web: http://www.cs.rug.nl/~wim

Abstract

The lazy caching algorithm of Afek, Brown, and Merrit (1993) is a protocol that allows the use of local caches with delayed updates. It results in a memory model that is not atomic (linearizable) but only sequentially consistent as defined by Lamport. In Distributed Computing **12** (1999), specifying and proving sequential consistency for the lazy caching algorithm was made into a benchmark for verification models. The present note contains such a specification and proof. It provides a simulation from the implementation to the abstract specification. The concrete verification only relies on the state space and the next-state relation. All behavioural aspects are treated in theories independent of the specific algorithm. The proofs of the underlying theories and of the concrete algorithm have been verified with the proof assistant PVS.

Key words: Sequential consistency, cache coherence, mechanical verification, simulation, weak fairness

1 Introduction

Lazy caching applies in a multiprocessor architecture that implements a shared memory. For the sake of performance, different processors may rely on local caches rather than inspecting or modifying the shared memory itself. Cache coherence would imply that the system's behaviour is logically indistinguishable from a shared memory with atomic updates, the so-called serial memory. This requirement, however, is under some circumstances inconvenient and stronger than necessary.

Sequential consistency is a weaker alternative. A memory is said to be *sequentially consistent* iff every behaviour xs of it allows a behaviour ys of the serial memory, such that, for every process p, the projections $xs|p$ and $ys|p$ are equal. In other words, the processes cannot distinguish the memory from a serial one. The difference becomes apparent when the system is equipped with a central clock or some other synchronization primitive. Then it turns out that, in a sequentially consistent memory, processes can read obsolete values, i.e., values for which another process has successfully completed a modification command. For many purposes, however, sequential consistency is good enough. Since strict synchronization hampers performance, a sequentially consistent memory may have better average performance than serial memory.

The lazy caching algorithm is an implementation of sequentially consistent memory proposed in [ABM93]. Inspired and introduced by the late Rob Gerth [Ger99], the algorithm was studied in various formalisms in [Bri99, Gra99,

JPZ99, JPR99, LD99, LLOR99]. In [Aro01], it was verified with the proof assistent PVS [OSRSC01] with a proof based on reordering of execution sequences using timestamps.

The present paper contains a refinement proof of the algorithm that has also been verified with PVS. Our specification of sequential consistency is a simplification of the one of [LLOR99]. We follow and extend [ABM93] in the choice of history variables.

We use a form of refinement calculus [Hes05a] based on [AL91], also inspired by TLA [Lam94]. The approach is related to UNITY [CM88] and linear temporal logic [MP92]. For instance, like in UNITY, we use ordinary assignments rather than primed variables to specify computation steps. Unlike TLA, but just as is usual in ordinary mathematics, the translation to a completely formal system is kept behind the scenes, but in our case it can be inspected by consulting the PVS proof script.

In [ABM93], the operations of writing and reading were split in a request for the action and a result or acknowledgement after the action. Gerth [Ger99] simplifies this by regarding writing and reading as single transitions. Since we want to model that the system to be implemented does not itself invent the addresses where it reads or writes, nor the values to be written, we follow [ABM93] and split every process in two parts: an environment that nondeterministically chooses where to read and write and what to write and a system component that executes these requests. Extending [ABM93], we even allow this environment concurrent read and write requests that can be answered asynchronously by the system components. Our problem is thus slightly more difficult than the problem considered in [ABM93, Aro01, Bri99, Gra99, JPZ99, JPR99, LD99, LLOR99].

After the introduction of history variables, the paper [ABM93] relies on the analysis of complete behaviours to prove sequential consistency. Our specification formalism and refinement approach enable us to avoid this. In this, our approach is similar to [LLOR99]. We extend the concrete specification with auxiliary actions to conform to the abstract specification. This is formalized by the concept of simulations [Hes06]. The special case used here is that of splitting simulations, a variation of extension with stuttering variables as defined in [LLOR99].

The final part of the correctness argument consists of a refinement mapping from the specification that is obtained by extending the concrete specification with auxiliary variables and auxiliary actions to the abstract specification. The proof that this is indeed a refinement mapping, has a safety part that is essentially trivial and a progress part that is critical. At this point, we use and extend the rules for "leads-to" introduced in [CM88], rather than relying on straight temporal reasoning as [LLOR99] does. Another difference with [LLOR99] is that our proof is complete in the sense that it is a report of a complete mechanical verification with the proof assistant PVS, whereas the proof in [LLOR99] is explicitly incomplete.

Verification with a proof assistant like PVS should be regarded as a method to completely and convincingly prove a mathematical assertion. During the

development of the proof, the tool assists by highlighting proof obligations and exposing faulty deductions. Since clumsy proof efforts are very time consuming (for the human user) and often fail, using a proof assistant forces us to strive for elegant proofs. After completion of the proof, the proof script in combination with the tool is a witness for the validity of the assertion and its proof.

Methodological contributions. We treat all behavioural aspects in theories that are independent of the algorithm under investigation. The verifications at the concrete level are only connected with the state space and the next-state relations, i.e., with single steps of the algorithm. In particular, we do not reorder the histories of the concrete algorithm as is done in [ABM93, Aro01].

Our specification of sequential consistency is a simplification of the one of [LLOR99]. We justify it by relating it to the verbal definition of sequential consistency. The lazy caching algorithm inspired us to introduce splitting simulations [Hes06]. It now serves as a first case study for their application. In this case, proving the progress properties is critical. Here we extend concepts and results of [CM88].

Overview. Section 2 contains the introduction to the specification formalism and to our use of temporal logic. In Section 3, we formalize the memory and its sequential consistency in an abstract specification *Kab*, which is closely related to the specification used by [LLOR99].

In Section 4, we introduce strict simulations and general simulations, and the splitting versions of them, to validate implementation relations. For the proofs of the nontrivial results of Section 4, we refer to the companion papers [Hes06, Hes05b]. Section 5 contains the lazy caching algorithm of [ABM93], modelled as a concrete specification K_0. We form a relation *Fca* between the state spaces of K_0 and the abstract specification *Kab*, and claim in Theorem 4 that *Fca* is a simulation from K_0 to *Kab*.

In Section 6, we extend the lazy caching algorithm with history variables to a specification K_1, and derive a host of invariants for it. In order to come closer to the abstract specification, we then add some more auxiliary variables and some auxiliary steps to obtain a specification K_2. The relationship from K_0 to K_2 is proved to be a splitting simulation.

In Section 7, we define and extend the relation "leads-to" of [CM88], and state a number of laws for it. In Section 8, we first restrict specification K_2 by means of the invariants to a specification K_3. We then prove that K_3 satisfies the progress conditions postulated for the abstract specification *Kab*. This enables us to define a refinement mapping from K_3 to *Kab*. By composition, this yields a simulation from K_0 to *Kab*. We finally prove that *Fca* is a simulation from K_0 to *Kab*, as claimed in Theorem 4, by proving that it contains this composition.

In Section 9, we briefly describe the mechanical verification, which consists of a hierarchy of PVS theories that proves the results of this paper and its companion [Hes06]. Conclusions are drawn in Section 10.

2 Specifications

In this section, we present our formalism for specifications, which is based on [AL91]. If X stands for the state space, predicates on X correspond to sets of states, relations over X correspond to possible state transformations, computations give rise to infinite sequences over X. A specification is a state machine over X with a supplementary property to specify progress.

2.1 Predicates, Subsets, Products, and Relations

A predicate (boolean function) on a set X is identified with the subset of X where the predicate holds. We can therefore identify conjunction ($\wedge$) and disjunction ($\vee$) with intersection ($\cap$) and union ($\cup$), respectively. Negation ($\neg$) is the same as complementation with respect to X. Implication is the set operation with $(U \Rightarrow V) = (\neg U \vee V)$. On the other hand, $U \subseteq V$ expresses that predicate U is stronger than predicate V, i.e., that $(U \Rightarrow V) = X$.

Just as in the language of the proof assistant PVS, square brackets are used to denote Cartesian products and the components of a Cartesian product are numbered from 1. So, e.g., if $x \in [A, B, C]$, then $x = (x_1, x_2, x_3)$ with $x_1 \in A$, etc.

A binary relation on a set X is identified with the set of pairs that satisfy the relation; this is subset of the Cartesian product $X^2 = [X, X]$. We write $\mathbf{1}$ for the identity relation of X. If A is a binary relation on X and Q is a predicate on X, its *weakest precondition* is defined by

$$wp(A, Q) = \{x \mid \forall\, y : (x, y) \in A \;\Rightarrow\; y \in Q\} \ .$$

A special case is $\mathit{disabled}(A) = wp(A, \emptyset)$.

2.2 Temporal Formulas

We write X^ω for the set of infinite sequences on X, which are regarded as functions $\mathbb{N} \to X$. For a sequence xs, we write $\mathit{Suf}(xs)$ to denote the set of its (infinite) suffixes. If P is a set of sequences, the sets $\Box P$ (always P), and $\Diamond P$ (sometime P) are defined by

$$\begin{array}{lcl} xs \in \Box P & \equiv & \mathit{Suf}(xs) \subseteq P \ , \\ \Diamond P & = & \neg\Box\neg P \ . \end{array}$$

So, $xs \in \Box P$ means that all suffixes of xs belong to P, and $xs \in \Diamond P$ means that xs has some suffix that belongs to P.

For $U \subseteq X$, we define the subset $[\![U]\!]$ of X^ω to consist of the sequences whose first element is in U. For a relation A on X, we define the subset $[\![A]\!]_2$ of X^ω to consist of the sequences that start with an A-transition. So we have

$$\begin{array}{lcl} xs \in [\![U]\!] & \equiv & xs(0) \in U \ , \\ xs \in [\![A]\!]_2 & \equiv & (xs(0), xs(1)) \in A \ . \end{array}$$

In temporal logic, these operators are usually kept implicit.

The weak fairness set $WF(A)$ of a relation A is defined to consist of the sequences that take infinitely many A transitions if A is in some suffix always enabled. Following Lamport [Lam94], we use the formal definition

$$WF(A) \quad = \quad \Box\Diamond[\![\, \mathit{disabled}(A)\,]\!] \quad \vee \quad \Box\Diamond[\![\, A\,]\!]_2 \ .$$

A sequence *ys* is defined to be a *stuttering* of a sequence *xs*, notation $xs \preceq ys$, iff *ys* can be obtained from *xs* by replacing its elements by positive iterations of them, so that $v = xs(n)$ is replaced by $v^{d(n)}$ for some function $d : \mathbb{N} \to \mathbb{N}_+$. For example, if, for a finite list *vs*, we write vs^ω to denote the sequence obtained by concatenating infinitely many copies of *vs*, the sequence $(aaabbbccb)^\omega$ is a stuttering of $(abbccb)^\omega$. See [Hes06], Section 6, for a formalization of $\preceq$.

A subset P of X^ω is called a *property* [AL91, Hes05a] iff it is insensitive to stutterings, i.e., if $(xs \in P) \equiv (ys \in P)$ whenever $xs \preceq ys$. If P is a property, then $\Box P$, and $\Diamond P$, and $\neg P$ are properties. The conjunction and disjunction of properties is a property. $[\![\, U\,]\!]$ is a property for every $U \subseteq X$. If A is a reflexive relation on X, then $\Box\,[\![\, A\,]\!]_2$ is a property. If A is irreflexive, then $\Diamond\,[\![\, A\,]\!]_2$ is a property. It follows that $WF(A)$ is a property for every irreflexive relation A.

If X has more than one element, not every subset of X^ω is a property. For example, the set $\Box\Diamond\,[\![\, \mathbf{1}\,]\!]_2$, which consists of the sequences that stutter infinitely often, is not a property.

2.3 Specifications and Programs

Following [AL91], a *specification* is defined to be a tuple $K = (X, Y, N, P)$ where X is the state space, $Y \subseteq X$ is the set of initial states, $N \subseteq X^2$ is the next-state relation and P is the supplementary property. Relation N is required to be reflexive in order to allow stutterings. P is a subset of the set X^ω of the infinite sequences of states, which is required to be a property.

We define an *initial execution* of K to be a sequence *xs* over X with $xs(0) \in Y$ and such that every pair of consecutive elements belongs to N. A *behaviour* of K is an infinite initial execution *xs* of K with $xs \in P$. We write $Beh(K)$ to denote the set of behaviours of K. It is easy to see that

$$Beh(K) = [\![\, Y\,]\!] \cap \Box[\![\, N\,]\!]_2 \cap P \ .$$

A state $x \in X$ is said to be *occurring* iff $x = xs(n)$ for some behaviour *xs* of K and some number n. A subset J of X is called an *invariant* of K iff J contains all occurring states. A subset J is called *inductive* for K if $Y \subseteq J$ and $J \subseteq wp(N, J)$. It is well known and easy to prove that every inductive subset of X is an invariant.

We use specifications to model concurrent systems with processes that act on shared variables and also have private variables. In this setting, a state of the system is given by the values of all variables and the state space X is the set of all states.

We declare shared variables with the keyword **var** and write them in type writer font. Private variables are slanted and declared with the keyword **privar**.

Outside of the programs, a private variable v of process p is denoted by $v.p$. Indeed, formally, a private variable is treated as a (modifiable) function from process identifiers to values. In the program, we use *self* to denote the identifier of the acting process.

The initial values of the variables are given at the declaration. The next-state relation N is given as a program in guarded command notation, where we keep the possibility of stuttering steps implicit. A construct of the form

(W) **whenever**
$[\!]\ U_i \ \rightarrow\ A_i$;
end

denotes a next-state relation that is the union of the identity relation $\mathbf{1}$ with the sets $A_i \cap [U_i, X]$. So, it is a nondeterminate choice between the guarded commands $U_i \rightarrow A_i$, which are taken atomically and repeatedly. A parallel composition of such constructs (W) denotes the union of their next-state relations. The difference with Dijkstra's **do od** notation is that the **do od** construct terminates when none of the guards hold, whereas (W) never terminates. When none of the guards hold, the construct (W) just blocks waiting for some other component to modify a guard. We omit the guard U_i when it equals the predicate *true*, i.e. the set X.

The supplementary property is given separately by means of some temporal logic formula.

2.4 Additional Notations

In the description of the algorithms, we use the following notations.

If T is a type, we write $T_\perp$ for the extension $T \cup \{\perp\}$ of type T with a new symbol $\perp$ that is used for "undefined" or `null`. We write T^* to denote the type of the finite lists (sequences) over T. We write ε to denote the empty list. We write $last(xs)$ for the last element of a nonempty finite list xs. We write $add(xs, x)$ for the command to append element x at the end of the finite list xs. For a nonempty list xs, we use the command $pop(xs)$ to return and remove the head of the list. The length of $xs \in T^*$ is denoted by $\#xs$. We write $xs(i)$ for the i-th element of list xs, starting from 0, and provided $i < \#xs$.

3 Specifications of Memory

A *memory* is a system with one shared array variable declared by:

type *Memory* = **array** *Address* **of** *Data* ;
var `memA` : *Memory* := `mem0` .

In a first approximation, the memory can be used by a system AuS of processes p that read and write at various addresses according to

$AuS.p$: **whenever**
$[\!]$ *choose* $a \in Address$; *choose* $d \in Data_\perp$;
if $d \neq \perp$ **then** $\texttt{memA}(a) := d$
else $d := \texttt{memA}(a)$ **end** ;
end .

Here, the **then** branch stands for writing into memory, and the **else** branch reads from memory. Recall from 2.3 that the **whenever** clause models a non-terminating program in which the guarded statements are executed atomically and repeatedly, whenever enabled. Autonomous system AuS is not very useful, since the processes are allowed to decide for themselves where to read and write and what to write.

3.1 The Environment and the Serial Memory

In order to model that the memory has to serve requests from clients that it does not control, we delegate such requests to an environment. Every client process p is split into two parts: an environment $Env.p$ that generates the requests, and a system component $CoS.p$ that executes them. These parts have the same private variables. We provide specifications for Env and CoS. The implementation consists of a concrete program that combined with Env implements the abstract combination of Env and CoS.

We declare private variables $addr$, ar, and val to hold the requests generated by the environment, and a private variable $result$ for the result of the read action.

privar $addr, ar : Address_\perp$;
privar $val, result : Data$.

Here, $addr \neq \perp$ means that there is a request to write value val at address $addr$. If $ar \neq \perp$, this means a request to read the value stored at address $addr$ and to assign it to $result$.

We first assume that the environment components are *serial* in the sense that they make no concurrent write and read requests. They are therefore modelled by

$EnvS.p$: **whenever**
$[\!]$ $addr = \perp \ \wedge \ ar = \perp \quad \rightarrow \quad$ *choose* $addr \in Address$, $val \in Data$;
$[\!]$ $addr = \perp \ \wedge \ ar = \perp \quad \rightarrow \quad$ *choose* $ar \in Address$;
end .

The serial memory system answers the write requests in `Wr` and the read requests in `Rd`, according to

$CoS.p$: **whenever**
`Wr` $[\!]$ $addr \neq \perp \quad \rightarrow \quad \texttt{memA}(addr) := val$; $addr := \perp$;
`Rd` $[\!]$ $ar \neq \perp \quad \rightarrow \quad result := \texttt{memA}(ar)$; $ar := \perp$;
end .

The progress condition is that all write and read requests are eventually answered:

(SF0) $\forall\, q : \Box \Diamond [\![\, \mathtt{addr}.q = \bot \,]\!]$,
(SF1) $\forall\, q : \Box \Diamond [\![\, ar.q = \bot \,]\!]$.

This concludes specification *SM* of the serial memory.

3.2 The Asynchronous Serial Memory

We prefer to allow a less deterministic environment that can make concurrent write and read requests, according to

Env.*p* : **whenever**
$[\![$ $addr = \bot \quad \rightarrow \quad$ *choose* $addr \in Address$, $val \in Data$;
$[\![$ $ar = \bot \quad \rightarrow \quad$ *choose* $ar \in Address$;
end .

The *asynchronous serial memory ASM* is the parallel composition of the environment components *Env*.*p* with the system components *CoS*.*p* for all *p*. We retain progress condition (SF0), but weaken condition (SF1) to

(AF1) $\forall\, q :\ \Box \Diamond [\![\, ar.q = \bot \,]\!] \quad \vee \quad \Box \Diamond [\![\, \mathtt{Wr}.q \,]\!]_2$.

This says that, for every process *q*, every write request is eventually answered, and that every read request is eventually answered unless the process itself writes infinitely often.

We regard the requests as asynchronous, and not as a part of the resulting operation. The memory is only serial with respect to the operations themselves. Indeed, it is debatable whether the word *serial* for this kind of memory is justified.

Note that in the system *SM* of Section 3.1, the conditions (SF1) and (AF1) are equivalent since writing has the precondition $addr.p \neq \bot$ which in *SM* implies $ar.p = \bot$.

3.3 Sequential Consistency

We now turn to distribution and sequential consistency. Recall that a memory is *sequentially consistent* iff, for every behaviour *xs* of it, there is a behaviour *ys* of the serial memory such that for every process *q* the projections of *xs* and *ys* to *q* are equal. Since we choose to allow the environment *Env* of 3.2, we decide to ignore the requests in these projections of *xs* and *ys*.

We formalize this version of sequential consistency by introducing auxiliary variables *loca*.*p* and *locs*.*p* to stand for the respective projections.

type $Action \;=\; [Address, Data, Bool]$;
privar $loca, locs : Action^* := \varepsilon$.

Reading a value d at an address a is represented by the action (a, d, false). Writing a value d at address a is represented by the action (a, d, true).

As announced above, the environment *Env* stays the same. The interaction with the environment is registered in *loca* and generated by

CoD.p : **whenever**
`Wr` ▯ $addr \neq \bot \quad \rightarrow \quad add(loca, (addr, val, true))$; $addr := \bot$;
`Rd` ▯ $ar \neq \bot \quad \rightarrow$
choose result $\in Data$; $add(loca, (ar, result, false))$; $ar := \bot$;
end .

The choice of *result* in `Rd` is nondeterminate, but is constrained by a hidden autonomous serial memory, the actions of which are registered in *locs*:

AuH.p : **whenever**
▯ *choose* $a \in Address$; $d \in Data_\bot$;
if $d \neq \bot$ **then** $\texttt{memA}(a) := d$; $add(locs, (a, d, true))$
else $d := \texttt{memA}(a)$; $add(locs, (a, d, false))$ **end** ;
end .

Sequential consistency requires that, for every process q, the limits of *loca.q* and *locs.q* when time goes to infinity, are equal. Since these lists change only by growing, this condition is equivalent to

$$i < \#loca.q \ \wedge \ i < \#locs.q \quad \Rightarrow \quad loca.q(i) = locs.q(i) \ ,$$
$$\lim \#loca.q = \lim \#locs.q \ .$$

In words, the lists are always equal as far as defined, and in the limit they have equal lengths (possibly ∞), for all q. We furthermore retain the progress conditions (SF0) and (AF1).

3.4 A Less Redundant Specification

The specification of Section 3.3 asks us to keep the local histories in two sets of growing lists that have to be equal as far as defined. We prefer to remove the redundancy and eliminate *loca* or *locs*. Since the externally visible things enter via *loca*, we eliminate *locs*. We do this by deciding that *locs.q* is always a prefix of *loca.q*, say of length *ser.q*. The guarded commands of *AuH.p* now become incrementations of *ser.p*. They are only enabled when $ser.p < \#loca.p$. The nondeterminacy of *AuH* is resolved by $loca.p(ser.p)$, the action to be included in *locs.p*. We thus replace *AuH* by a kind of delayed serialization:

privar *ser* : *Nat* := 0 ;
AuD.p : **whenever**
`WrD` ▯ $ser < \#loca \ \wedge \ loca(ser)_3 \quad \rightarrow$
$\texttt{memA}(loca(ser)_1) := loca(ser)_2$; *ser*`++` ;
`RdD` ▯ $ser < \#loca \ \wedge \ \neg\, loca(ser)_3 \ \wedge \ \texttt{memA}(loca(ser)_1) = loca(ser)_2 \quad \rightarrow$
ser`++` ;
end .

The boolean $loca(ser)_3$ in the guards of `WrD` and `RdD` determines whether the action is a write action or a read action. The third conjunct of the guard of `RdD` is needed to keep *locs* a prefix of *loca*, and serves to justify the choice of *result* in `Rd`.

To ensure consistent choices in `Rd` or, formally, to ensure that *loca* and *locs* are equal in the limit, we need to impose the progress condition:

(AF2) $\quad \forall\, q, n :\ \Box\,(\llbracket\, \#loca.q \geq n \,\rrbracket \ \Rightarrow\ \Diamond \llbracket\, ser.q \geq n \,\rrbracket)\ .$

Note that this is only a specification. The implementation is responsible for consistent choices in `Rd` such that (AF2) can be established.

To summarize, our abstract specification $Kab = (X, Y, N, P)$ consists of the state space X and the initial set Y defined by the declarations of the shared variable `memA` and the private variables *addr*, *val*, *ar*, *result*, and *loca*. The next-state relation N is given as the union of the programs $Env.p$ and $CoD.p$ and $AuD.p$, for all $p \in Process$. The supplementary property P is given as the conjunction of the properties (SF0), (AF1), and (AF2).

Only the private variables *addr*, *val*, *ar*, and *result* are considered visible. The variables `memA`, *loca*, *ser* are invisible specification variables that need not be implemented.

Apart from our separation between system and environment, this specification is essentially equivalent with specification *SeqDB1* of sequential consistency in [LLOR99], Figure 12. The remaining difference is that our lists *loca* are only traversed by *ser* and not dequeued as in [LLOR99]. Since *loca* and *ser* are auxiliary variables, this is only a matter of convenience, not of space complexity.

Remarks. Strictly speaking, sequential consistency should be defined as in Section 3.3, say with specification *Ksc* obtained from *Kab* by using AuH instead of AuD. The arguments given above show that *Kab* implements *Ksc*. This is enough for our purposes. It is not very difficult to prove that, conversely, *Ksc* implements *Kab*. This would solve the question raised by Meritt in [Mer99] (p. 56). Since we do not need it, we leave this to the interested reader.

4 Simulations

Before presenting the lazy-caching implementation of specification *Kab* of Section 3, we describe the formalism to express and prove correctness of implementations. The formalism we employ is a variation of the concepts introduced in [AL91]. The idea is that every behaviour of the implementing specification, say K, should correspond in a certain sense to some behaviour of the abstract specification, say L.

We use simulations to formalize this. There are several ways to prove that a relation is a simulation. The easiest way is to use that every refinement mapping induces a simulation. It is well known, however, that refinement mappings are often too specific to prove some implementation relation. Another way is to extend the state space with history variables [AL91], as formalized in the

concept of forward simulations. In practice, the traditional way to deal with liveness (progress) for forward simulations is inconvenient. In our specific case, we need to add not only auxiliary variables but also auxiliary steps to match the concrete specification with the abstract one. We therefore propose splitting simulations as an alternative to forward simulations.

4.1 Notations to Relate Specifications

If F is a relation between sets X and Y, we write F^ω for the relation between the sets of infinite sequences X^ω and Y^ω that is given by

$$(xs, ys) \in F^\omega \quad \equiv \quad (\forall\, i : (xs(i), ys(i)) \in F)\ .$$

If F is a binary relation between sets X and Y, and G is a binary relation between Y and Z, the relational composition $(F; G)$ is defined to consist of the pairs (x, z) such that there exists y with $(x, y) \in F$ and $(y, z) \in G$.

When we have to consider more than one specification, the components of a specification $K = (X, Y, N, P)$ are denoted $\mathit{states}(K) = X$, $\mathit{start}(K) = Y$, $\mathit{step}(K) = N$ and $\mathit{prop}(K) = P$.

4.2 Simulations

A *strict simulation* F from a specification K to a specification L (notation $F : K \rightarrowtriangle L$) is defined to be a relation F between $\mathit{states}(K)$ and $\mathit{states}(L)$ such that, for every $xs \in \mathit{Beh}(K)$, there exists $ys \in \mathit{Beh}(L)$ with $(xs, ys) \in F^\omega$.

This definition of strict simulation requires that the abstract behaviour matches the concrete one step by step. Since the specifications allow stuttering, this definition allows the abstract behaviour to stutter while the concrete behaviour is doing "internal steps". It does not allow the abstract behaviour to do additional steps when the concrete behaviour happens to proceed. In the present case, this strictness turns out to be inconvenient. Indeed, we need to allow that the concrete behaviour occasionally takes fewer steps than the abstract behaviour [Lam89]. We do this by defining (nonstrict) simulations in the following way.

A relation F between the state spaces of specifications K and L is defined to be a *simulation* [Hes06] from K to L, notation $F : K \twoheadrightarrow\!\!\!\!\triangleright\ L$, if for every $xs \in \mathit{Beh}(K)$ there exists a pair $(xt, ys) \in F^\omega$ with $xs \preceq xt$ and $ys \in \mathit{Beh}(L)$.

According to this definition, it is possible that the concrete behaviour xs takes less steps than needed by the abstract behaviour. In fact, it suffices that the abstract behaviour matches the steps of a slowed-down "virtual" concrete behaviour xt.

It is clear that every strict simulation is a simulation since one can choose $xt = xs$. Conversely, however, not all simulations are strict. It is easy to prove that the relational composition of strict simulations is a strict simulation. Using an adequate definition of relation $\preceq$, it is also easy to prove that the relational composition of simulations is a simulation [Hes05b].

4.3 Refinement Mappings and Invariant Restrictions

If K and L are specifications, a function $f : \mathit{states}(K) \to \mathit{states}(L)$ is called a *refinement mapping* [AL91] from K to L iff $f(x) \in \mathit{start}(L)$ for every $x \in \mathit{start}(K)$, and $(f(x), f(x')) \in \mathit{step}(L)$ for every pair $(x, x') \in \mathit{step}(K)$, and $f \circ xs \in \mathit{prop}(L)$ for every $xs \in \mathit{Beh}(K)$. In this situation we regard L as an abstract specification implemented by a concrete specification K.

It is easy to see that, if f is a refinement mapping from K to L, the graph of f is a strict simulation $K \rightarrowtriangle L$. We therefore also regard a refinement mapping f as a strict simulation $f : K \rightarrowtriangle L$.

If J is an invariant of a specification $K = (X, Y, N, P)$, we can restrict the state space to the set J and thus form the restricted specification $L = (J, Y, N \cap J^2, P \cap J^\omega)$. Clearly, every behaviour of K is also a behaviour of L. Therefore, the identity relation $\mathbf{1}$ of J regarded as a relation between X and J is a strict simulation $\mathbf{1} : K \rightarrowtriangle L$. Specification L is called the *invariant restriction* $\mathit{res}(K, J)$.

4.4 Forward Simulations

A third easy method to prove that one specification simulates another is by starting at the beginning and constructing the corresponding behaviour in the other specification inductively. This idea is formalized in forward simulations [HHS86, Hes05a, LV95, Mil71], defined as follows.

A relation F between $\mathit{states}(K)$ and $\mathit{states}(L)$ is called a *forward simulation* from specification K to specification L iff

(F0) For every $x \in \mathit{start}(K)$, there is $y \in \mathit{start}(L)$ with $(x, y) \in F$.
(F1) For every pair $(x, y) \in F$ and every x' with $(x, x') \in \mathit{step}(K)$, there is y' with $(y, y') \in \mathit{step}(L)$ and $(x', y') \in F$.
(F2) Every initial execution ys of L with $(xs, ys) \in F^\omega$ for some $xs \in \mathit{Beh}(K)$ satisfies $ys \in \mathit{prop}(L)$.

The following lemma [Hes05a] expresses soundness of forward simulations:

Lemma 1 *Every forward simulation F from a specification K to a specification L is a strict simulation $F : K \rightarrowtriangle L$.*

A forward simulation $F : K \rightarrowtriangle L$ is called a *history extension* iff it is the converse of a refinement mapping $L \rightarrowtriangle K$. Usually, the state space of K is spanned by some variables, the state space of L is spanned by the same variables together with some auxiliary variables, and the refinement mapping from L to K is the projection function that forgets the values of the auxiliary variables. Roughly speaking, condition (F0) is a matter of consistent initialization, condition (F1) says that the steps of K are faithfully represented by L, and condition (F2) says that no additional progress conditions are imposed.

4.5 Splitting Simulations

In practice, condition (F2) of section 4.4 is inconvenient since all fairness conditions of the specifications K and L are accumulated before the supplementary properties are compared. There even are cases where an obviously sound extension of a specification with a history variable is not a forward simulation, cf. [Hes06]. It is therefore preferable to compare the fairness conditions of K and L one by one.

A *splitting* of specification K is a family of relations $(i \in \mathbb{N} : A.i)$ such that

$$\begin{aligned} &\mathit{step}(K) = \mathbf{1} \cup (\bigcup i \in \mathbb{N} : A.i) \ , \\ &\mathit{prop}(K) = (\bigcap i \in \mathbb{N} : i \neq 0 : \mathit{WF}(A.(i))) \ . \end{aligned}$$

So, the alternatives $A.i$ are subrelations of $\mathit{step}(K)$ and all nonstuttering steps of K belong to some alternative $A.i$. The positive alternatives are treated with weak fairness.

Let K and L be specifications. A *strict splitting simulation* from K to L is a relation F between the state spaces of K and L such that condition (F0) of section 4.4 holds and that there exist splittings $(i \in \mathbb{N} : A.i)$ and $(i \in \mathbb{N} : B.i)$ of K and L, respectively, that satisfy

(F1s) If $(x, y) \in F$ and $(x, z) \in A.i$, there is w with $(z, w) \in F$ and $(y, w) \in B.i$.
(F2s) If $(x, y) \in F$ and $i > 0$ and $x \in \mathit{disabled}(A.i)$, then $y \in \mathit{disabled}(B.i)$.

In [Hes06], soundness of strict splitting simulations is proved, as expressed by

Theorem 2 *Every strict splitting simulation is indeed a strict simulation.*

The proof of this result is easy when one imposes the additional assumption that the alternatives $A.i$ with $i > 0$ are pairwise disjoint. We do not want to make this assumption. The proof in [Hes06] therefore requires a kind of scheduler.

When we need to augment the concrete specification with auxiliary steps to match the abstract specification, the simulation cannot be strict anymore. We therefore define general splitting simulations in the following way.

A *splitting simulation* from K to L is defined to be a relation F between the state spaces of K and L such that condition (F0) of section 4.4 holds and that K and L have splittings $(i \in \mathbb{N} : A.i)$ and $(i \in \mathbb{N} : B.i)$, respectively, such that condition (F1s) holds and:

(F2ns) If $(x, y) \in F$ and $i > 0$ and $x \in \mathit{disabled}(A.i)$, then $y \in \mathit{disabled}(B.i)$ or there exists w with $(y, w) \in B.i$ and $(x, w) \in F$.

There are two principal possibilities to satisfy condition (F2ns). Let alternative i be called *conservative* iff, for every pair $(x, y) \in F$, we have that $x \in \mathit{disabled}(A.i)$ implies $y \in \mathit{disabled}(B.i)$ as in (F2s). Let alternative i be called *stuttering* iff, for every pair $(x, y) \in F$ and every w with $(y, w) \in B.i$, we have that $(x, w) \in F$. It is easy to see that (F2ns) holds if every alternative $i > 0$ is conservative or stuttering. Clearly, however, condition (F2ns) is weaker than this disjunction. Note that (F2ns) can hold while $A.i$ is empty (i.e. absent) and $B.i$ is nonempty.

The next result of [Hes06] expresses soundness of splitting simulations:

Theorem 3 *Every splitting simulation is indeed a simulation.*

Remark. In [Hes06], we also allow splittings where some alternatives have strong fairness requirements. We do not need this here.

5 The Lazy Caching Algorithm

Lazy caching is a sequentially consistent memory system introduced in [ABM93]. The idea is that processes should delay communication as much as possible without violating sequential consistency. The algorithm uses the private variables *addr*, *val*, *ar*, and *result*, which are visible in the specification of section 3, and the new private varibles declared by

$$\begin{array}{l}\textbf{type}\ \ Pair\ \ =\ \ [Address, Data]\ ;\\ \textbf{privar}\ \ out : Pair^*\ :=\ \varepsilon\ ;\\ \textbf{privar}\ \ cache : \textbf{array}\ Address\ \textbf{of}\ Data_\perp\ :=\ (\lambda a : \perp)\ .\end{array}$$

The queues *out* serve for communication from the processes towards memory. The variables *cache* hold the local caches. Values $\perp$ in the caches represent absence of values. The algorithm uses the shared variables declared by

$$\begin{array}{l}\textbf{var}\ \ \texttt{memC} : Memory\ :=\ \texttt{mem0}\ ;\\ \textbf{var}\ \ \texttt{in} : \textbf{array}\ Process\ \textbf{of}\ Action^*\ :=\ (\lambda q : \varepsilon)\ .\end{array}$$

Variable `memC` serves as a central memory. We distinguish `memC` from `memA` to allow different update moments. Array `in` holds queues for the communication from memory towards the processes.

The implementation consists of the system components

$$\begin{array}{lll}LC.p: & \multicolumn{2}{l}{\textbf{whenever}}\\ \texttt{Wr} & [\!] & addr \neq \perp \ \ \rightarrow\ \ add(out, (addr, val))\ ;\ \ addr := \perp\ ;\\ \texttt{Rd} & [\!] & inGuard\ \ \rightarrow\ \ result := cache(ar)\ ;\ \ ar := \perp\ ;\\ \texttt{MW} & [\!] & out \neq \varepsilon\ \ \rightarrow\\ & & \quad (a, d) := pop(out)\ ;\ \ \texttt{memC}(a) := d\ ;\\ & & \quad \textbf{for all}\ \ q\ \ \textbf{do}\ \ add(\texttt{in}(q), (a, d, q = self))\ \ \textbf{end}\ ;\\ \texttt{CU} & [\!] & \texttt{in}(self) \neq \varepsilon\ \ \rightarrow\ \ (a, d, b) := pop(\texttt{in}(self))\ ;\ \ cache(a) := d\ ;\\ \texttt{Mrq} & [\!] & ar \neq \perp \wedge \texttt{cache}(ar) = \perp\ \ \rightarrow\ \ add(\texttt{in}(self), (ar, \texttt{memC}(ar), false))\ ;\\ \texttt{MR} & [\!] & choose\ a \in Address\ ;\ \ add(\texttt{in}(self), (a, \texttt{memC}(a), false))\ ;\\ \texttt{CI} & [\!] & choose\ a \neq ar\ ;\ \ cache(a) := \perp\ ;\\ & \multicolumn{2}{l}{\textbf{end}\ .}\end{array}$$

As before, the alternatives `Wr` and `Rd` are the actions of writing and reading. The guard for reading is

$$\begin{array}{l}inGuard:\\ \quad ar \neq \perp\ \wedge\ cache(ar) \neq \perp\ \wedge\ out = \varepsilon\ \wedge\ (\forall\, i :: \neg\texttt{in}(self)(i)_3)\ .\end{array}$$

Firstly, the reading address must have been chosen and its value must be available. For the sake of sequential consistency, it is required that the process's *out*-queue is empty and that its `in`-queue contains no urgent items. An item in the `in`-queue is urgent when it corresponds to a write action of the process itself. Urgency is indicated by *true* at the third components of the `in`-items, while it was a star (*) in [ABM93, Ger99].

Remark. One may propose to weaken *inGuard* by only requiring that address a does not occur in $out.p$ or in $\mathtt{in}(p)$ with $\mathtt{in}(p)_3$. This weakening is incorrect, as is shown by the following scenario. Let `x` and `y` be locations with initial values 0. Let A and B be processes with the programs

$$\begin{array}{ll} A: & \mathtt{x} := 1\ ; \quad \mathit{read}(\mathtt{y})\ . \\ B: & \mathtt{y} := 1\ ; \quad \mathit{read}(\mathtt{x})\ . \end{array}$$

Sequential consistency requires that one of the two assignments takes place before the other assignment and therefore before both read statements. Therefore, at least one of the processes reads a value 1. The proposed weakening, however, would allow that both processes read zeroes. □

The alternative `MW` (memory-write) stands for the delayed action of transferring a queued element to the memory `memC` and to all `in`-queues. An item is marked as urgent only in the `in`-queue of the writing process itself. The memory `memC` is inspected in the memory-read alternatives `Mrq` and `MR`. In the `Mrq`, the process asks for the value of its requested address. The local caches are modified in the cache update command `CU`. The cache invalidate command `CI` models that elements can be deleted from the cache. The condition $a \neq ar$ is needed to be able to guarantee progress for reading.

We assume that weak fairness holds for the commands `Wr`, `Rd`, `MW`, `CU`, and `Mrq` of all processes q. This means that, whenever one of these commands is continuously enabled for some process q from some point onward in a behaviour, it is eventually taken. More precisely, we require the fairness condition:

$$\begin{array}{ll} \text{(CF0)} & \forall\, q \in \mathit{Process}:\ \mathit{WF}(\mathtt{Wr}.q)\ \wedge\ \mathit{WF}(\mathtt{Rd}.q)\ \wedge\ \mathit{WF}(\mathtt{MW}.q) \\ & \qquad\qquad \wedge\ \mathit{WF}(\mathtt{CU}.q)\ \wedge\ \mathit{WF}(\mathtt{Mrq}.q)\ . \end{array}$$

Note that this is indeed a property since the five commands are all irreflexive (i.e. change something, when enabled).

We need not postulate fairness for `MR` and `CI`. Indeed, command `Mrq` was introduced to postulate $\mathit{WF}(\mathtt{Mrq}.q)$ without imposing fairness for `MR`. One may notice that fairness for `Mrq` and `Rd` is not needed in e.g. [LLOR99] (Fig. 6) since there the read request is not separated from its response.

To summarize, the implementation is formalized as the specification $K_0 = (X_0, Y_0, N_0, P_0)$ with the state space X_0 and initial set Y_0 given by the declarations of the variables `memC`, `in`, *addr*, *val*, *ar*, *result*, *cache*, *out*. The next-state relation N_0 is given as the union of the programs $\mathit{Env}.p$ and $\mathit{LC}.p$ for all processes p. The supplementary property P_0 is (CF0). The challenge is to prove that K_0 implements specification *Kab* of Section 3.4.

Since the visible variables of the specification are the private variables *addr*, *val*, *ar*, and *result*, we define the binary relation *Fca* between the state spaces of K_0 and *Kab* to consist of the pairs of states (x, y) such that

$$\forall\, q \in \mathit{Process} :\ x.\mathit{addr}.q = y.\mathit{addr}.q \ \wedge\ x.\mathit{val}.q = y.\mathit{val}.q \\ \wedge\quad x.\mathit{ar}.q = y.\mathit{ar}.q \ \wedge\ x.\mathit{result}.q = y.\mathit{result}.q\ .$$

Here, we use x to refer to a state of implementing system K_0 and we regard *addr*, *val*, *ar*, and *result*, as fields of x that hold functions from processes to values. Similarly, y refers to a state of the abstract system *Kab*, which also has such fields. The states x and y are kept implicit in the discussion of the algorithm below. They do appear, however, in the PVS modelling of the algorithm. Our main result is:

Theorem 4 *Relation Fca is a simulation from K_0 to Kab.*

The proof of this result is the contents of the remainder of this paper. Note that relation *Fca* only relates the visible variables *addr*, *val*, *ar*, and *result*. The only steps that modify these variables are *Env*, `Wr`, and `Rd`. The steps `MW`, `CU`, `Mrq`, `MR`, `CI` of K_0 are internal. The simulation in the Theorem is nonstrict to give time for the delayed actions `WrD` and `RdD`. Since it is irrelevant, however, when precisely these delayed actions happen, it is not unlikely that *Fca* is even a strict simulation from K_0 to *Kab*.

The strong condition *inGuard*, needed for sequential consistency, has the effect that algorithm K_0 does not guarantee separate progress for reading as expressed in formula (SF1) of Section 3.1. It is not difficult to modify the algorithm in such a way that (SF1) holds, but in our view this would complicate the algorithm and its proof unnecessarily.

6 Auxiliary Variables for Lazy Caching

In order to prove that the lazy caching algorithm *LC* is sequentially consistent, i.e., that relation *Fca* is a simulation from K_0 to *Kab*, we extend specification K_0 with auxiliary variables, primarily with history variables. Most of the history variables of 6.1 and 6.2 were introduced in [ABM93] and were also used in [Aro01].

6.1 Propagation of time stamps

We introduce a shared history variable

$$\textbf{var}\ \ \texttt{time} : \mathit{Nat} := 0$$

to serve as a time stamp of the version of `memC`. The `time` is broadcast in `MW` via the `in`-queues. The declaration of `in` is therefore changed into

$$\textbf{type}\ \ \mathit{Quad}\ \ =\ \ [\mathit{Address}, \mathit{Data}, \mathit{Bool}, \mathit{Nat}]\ ; \\ \textbf{var}\ \ \texttt{in} : \textbf{array}\ \mathit{Process}\ \textbf{of}\ \mathit{Quad}^*\ := (\lambda q : \varepsilon)\ .$$

Locally, the `time` is recorded in the private history variables *lard* (last read), *alt* (alternative), and *sno* (a list of sequence numbers), as declared in

$$\textbf{privar}\ \ lard, alt : Nat := 0\ ;$$
$$\textbf{privar}\ \ sno : Nat^* := \varepsilon\ .$$

We retain the guarded commands `Wr` and `CI` of LC, and replace `Rd`, `MW`, `CU`, `Mrq`, and `MR` as follows.

$$\begin{array}{ll}
\mathtt{Rd}_1 & [\!] \ \ inGuard \ \ \rightarrow \\
 & \quad result := cache(ar)\ ; \quad add(sno, lard)\ ; \quad ar := \bot \\
\mathtt{MW}_1 & [\!] \ \ out \neq \varepsilon \ \ \rightarrow \\
 & \quad (a,d) := pop(out)\ ; \quad \mathtt{memC}(a) := d\ ; \\
 & \quad add(sno, \mathtt{time})\ ; \quad \mathtt{time\ ++}\ ; \quad alt := \mathtt{time}\ ; \\
 & \quad \textbf{for all}\ \ q\ \ \textbf{do}\ \ add(\mathtt{in}(q), (a, d, q = self, \mathtt{time}))\ \ \textbf{end}\ ; \\
\mathtt{CU}_1 & [\!] \ \ \mathtt{in}(self) \neq \varepsilon \ \ \rightarrow \\
 & \quad (a,d,b,n) := pop(\mathtt{in}(self))\ ; \\
 & \quad cache(a) := d\ ; \quad lard := n \\
\mathtt{Mrq}_1 & [\!] \ \ ar \neq \bot \wedge \mathtt{cache}(ar) = \bot \ \ \rightarrow \\
 & \quad add(\mathtt{in}(self), (ar, \mathtt{memC}(ar), false, \mathtt{time}))\ ; \\
\mathtt{MR}_1 & [\!] \ \ choose\ a \in Address\ ; \quad add(\mathtt{in}(self), (a, \mathtt{memC}(a), false, \mathtt{time}))\ .
\end{array}$$

The paper [ABM93] now provides some important invariants. The private variables *lard* get their values via the fourth components of the elements of the `in`-queues. This leads to the invariants

$$\begin{array}{ll}
\text{(Iq0)} & i < \#\mathtt{in}(q) \ \ \Rightarrow \ \ \mathtt{in}(q)(i)_4 \leq \mathtt{time}\ , \\
\text{(Iq1)} & lard.q \leq \mathtt{time}\ , \\
\text{(Iq2)} & i \leq j < \#\mathtt{in}(q) \ \ \Rightarrow \ \ \mathtt{in}(q)(i)_4 \leq \mathtt{in}(q)(j)_4\ , \\
\text{(Iq3)} & i < \#\mathtt{in}(q) \ \ \Rightarrow \ \ lard.q \leq \mathtt{in}(q)(i)_4\ .
\end{array}$$

We need to complement these invariants by showing that the sequence $\mathtt{in}(q)(\_)_4$ fills the range from $lard.q$ to `time`, as expressed in the invariants

$$\begin{array}{ll}
\text{(Iq4)} & \mathtt{in}(q) \neq \varepsilon \ \ \Rightarrow \ \ last(\mathtt{in}(q))_4 = \mathtt{time}\ , \\
\text{(Iq5)} & i + 1 < \#\mathtt{in}(q) \ \ \Rightarrow \ \ \mathtt{in}(q)(i+1)_4 \leq \mathtt{in}(q)(i)_4 + 1\ , \\
\text{(Iq6)} & \mathtt{in}(q) = \varepsilon \ \ \Rightarrow \ \ lard.q = \mathtt{time}\ , \\
\text{(Iq7)} & \mathtt{in}(q) \neq \varepsilon \ \ \Rightarrow \ \ \mathtt{in}(q)(0)_4 \leq lard.q + 1\ .
\end{array}$$

Actually, we only need (Iq7), but the other three are used to prove the invariance of (Iq7).

The private sequences $sno.q$ serve as private sequence numbers. It is easy to see that they are ascending and satisfy the invariants

$$\begin{array}{ll}
\text{(Jq0)} & i < \#sno.q \ \ \Rightarrow \ \ sno.q(i) \leq \mathtt{time}\ , \\
\text{(Jq1)} & i < j < \#sno.q \ \ \Rightarrow \ \ sno.q(i) \leq sno.q(j)\ .
\end{array}$$

Preservation of (Jq0) follows from (Iq1). Preservation of (Jq1) under reading follows from the following inequality, a variation of which occurs in [ABM93]:

(Jq2) $\quad sno.q \neq \varepsilon \quad \wedge \quad lard.q < last(sno.q) \quad \Rightarrow \quad last(sno.q) < alt.q$,
(Jq3) $\quad lard.q < alt.q$
$\quad \Rightarrow \quad (\exists\, i : i < \#\mathtt{in}(q) \ \wedge \ \mathtt{in}(q)(i)_3 \ \wedge \ \mathtt{in}(q)(i)_4 = alt.q)$.

The ugly invariant (Jq3) implies that the last conjunct of *inGuard* guarantees that every reading process has $alt \leq lard$.

Proof structure. The predicates (Iq0) and (Iq4) are inductive. Preservation of (Iq1) follows from (Iq0); preservation of (Iq2) from (Iq0); preservation of (Iq3) from (Iq1) and (Iq2); preservation of (Iq5) from (Iq4); preservation of (Iq6) from (Iq4); preservation of (Iq7) from (Iq5) and (Iq6); preservation of (Jq0) from (Iq1); preservation of (Jq1) from (Jq0), (Jq2), (Jq3); preservation of (Jq2) from (Iq3); preservation of (Jq3) from (Iq3).

6.2 Propagation of data

In order to follow the data stream and the write actions during the computation we introduce shared history variables

var $\mathtt{hm} : Memory^* := add(\varepsilon, \mathtt{memO})$;
var $\mathtt{pm} : [Process, Nat]^* := \varepsilon$.

Variable `hm` holds the consecutive values of `memC`. It is initialized with the singleton list that only contains the initial value `memO` of `memC` and `memA`. When process p writes a value into `memC`, the list `pm` is extended with the pair (p, i) where i is the index of the time stamp in $sno.p$. Since `memC` is modified only in `MW`, we need only to provide a new version of `MW`:

$\mathtt{MW}_2 \quad [\!] \quad out \neq \varepsilon \quad \rightarrow$
$\quad (a, d) := pop(out)$; $\quad \mathtt{memC}(a) := d$;
$\quad add(sno, \mathtt{time})$; $\quad \mathtt{time}\ \mathtt{++}$; $\quad alt := \mathtt{time}$;
$\quad$ **for all** q **do** $add(\mathtt{in}(q), (a, d, q = self, \mathtt{time}))$ **end** ;
$\quad add(\mathtt{hm}, \mathtt{memC})$; $\quad add(\mathtt{pm}, (self, \#sno - 1))$.

It is clear that we have the invariants

(Kq0) $\quad \#\mathtt{hm} = \mathtt{time} + 1$,
(Kq1) $\quad last(\mathtt{hm}) = \mathtt{memC}$.

We now follow the data by proving the invariants

(Kq2) $\quad (a, d, b, n) = \mathtt{in}(q)(i) \quad \Rightarrow \quad d = \mathtt{hm}(n)(a)$,
(Kq3) $\quad lard.q < n < \#\mathtt{hm} \quad \wedge \quad \mathtt{hm}(n-1)(a) \neq \mathtt{hm}(n)(a)$
$\quad \Rightarrow \quad (\exists\, i : \mathtt{in}(q)(i) = (a, \_, \_, n) \ \wedge \ (\forall\, j : j < i \ \Rightarrow \ \mathtt{in}(q)(j)_4 < n))$.

Predicate (Kq2) expresses that the elements of the `in`-queues faithfully represent values that occurred at the moment indicated by the time stamp. Predicate (Kq3) expresses that every modification of the memory, if not yet dealt with, is represented in the `in`-queue and precedes any other occurrences of the same time stamp. Note that we use underscores as wild cards. As a reviewer remarked,

the last conjunct of (Kq3) is redundant because of (Iq2). This redundancy makes the mechanical proof somewhat simpler, since it allows us to eliminate the argument that a nonempty set of natural numbers has a least element.

We now can prove the first main invariant, which expresses that the value in the cache corresponds to the value in the memory at "the time" of *lard*:

(Kq4) $\quad cache.q(a) = \bot \quad \vee \quad cache.q(a) = \mathtt{hm}(lard.q)(a)$.

This predicate is threatened only by the modifications of the *cache* and *lard* in command `CU`. If $cache.q(a)$ is modified in `CU`, preservation of (Kq4) follows from (Kq2). If $cache.q(a)$ is not modified, but $lard.q$ is modified, then $lard.q$ is incremented with 1 by (Iq3) and (Iq7). Preservation of (Kq4) then follows from (Kq3) with $n := lard.q + 1$.

With respect to list `pm` we have the easy invariants

(Kq5) $\quad \#\mathtt{pm} = \mathtt{time}$,
(Kq6) $\quad n < \#\mathtt{pm} \quad \wedge \quad (q, i) = \mathtt{pm}(n) \quad \Rightarrow \quad i < \#sno.q \quad \wedge \quad sno.q(i) = n$.

Proof structure. Predicates (Kq0) and (Kq5) are inductive. Preservation of (Kq1) follows from (Kq0); preservation of (Kq2) from (Iq0), (Kq0), (Kq1); preservation of (Kq3) from (Iq0), (Iq3), (Kq0), (Kq1); preservation of (Kq4) from (Iq1), (Iq3), (Iq7), (Kq0), (Kq2), (Kq3); preservation of (Kq6) from (Kq5).

6.3 Recording local history

We now deviate from [ABM93, Aro01]. In order to prove that the lazy caching algorithm implements specification *CoD*, we introduce the private specification variables *loca* in Section 3 as history variables. We postpone the treatment of serialization. The variables *loca* are modified in the alternatives `Wr` and `Rd` in the same way as in specification *CoD*:

$\mathtt{Wr}_2 \quad [\!] \quad addr \neq \bot \quad \rightarrow$
$\quad add(out, (addr, val))$; $\quad addr := \bot$;
$\quad add(loca, (addr, val, true))$;
$\mathtt{Rd}_2 \quad [\!] \quad inGuard \quad \rightarrow$
$\quad result := cache(\mathrm{ar})$; $\quad add(loca, (\mathrm{ar}, cache(\mathrm{ar}), false))$;
$\quad add(sno, lard)$; $\quad ar := \bot$.

It is easy to see that the length of *loca* is always the sum of the lengths of *sno* and *out*. If *loca* holds a write action, the sequence *sno* increases or will increase at that index. These facts are captured in the invariants

(Lq0) $\quad \#loca.q = \#sno.q + \#out.q$,
(Lq1) $\quad i < k < \#sno.q \quad \wedge \quad loca.q(i)_3 \quad \Rightarrow \quad sno.q(i) < sno.q(k)$.

In order to prove (Lq1), we introduce the invariants

(Lq2) $\quad i < \#sno.q \quad \wedge \quad loca.q(i)_3 \quad \Rightarrow \quad sno.q(i) < alt.q$,
(Lq3) $\quad alt.q \leq \mathtt{time}$.

The queues $out.q$ transfer write commands from process q to memory, as expressed in (Lq4). We introduce the invariant (Lq5) to express that actions receive a sequence number that relates them to the global history as recorded in the list `hm`. The effects of the write actions on the global history are expressed in (Lq6).

$$\begin{array}{ll}
\text{(Lq4)} & i < \#out.q \quad \wedge \quad out.q(i) = (a, d) \\
& \Rightarrow \quad loca.q(i + \#sno.q) = (a, d, true) \ , \\
\text{(Lq5)} & i < \#sno.q \quad \wedge \quad n = sno.q(i) \quad \wedge \quad loca.q(i) = (a, d, false) \\
& \Rightarrow \quad \mathtt{hm}(n)(a) = d \ , \\
\text{(Lq6)} & i < \#sno.q \quad \wedge \quad n = sno.q(i) \quad \wedge \quad loca.q(i) = (a, d, true) \\
& \Rightarrow \quad \mathtt{hm}(n+1) = (\mathtt{hm}(n) \ \textbf{with} \ (a) := d) \ .
\end{array}$$

We also need uniqueness of the writing sequence numbers, as expressed in

$$\begin{array}{ll}
\text{(Lq7a)} & i < \#sno.q \quad \wedge \quad loca.q(i)_3 \quad \wedge \quad j < \#sno.r \quad \wedge \quad loca.r(j)_3 \\
& \wedge \quad sno.q(i) = sno.r(j) \quad \Rightarrow \quad q = r \quad \wedge \quad i = j \ .
\end{array}$$

Since an invariant with four free variables (i, j, q, r) requires complicated proofs, we introduced the auxiliary variable `pm` in section 6.2. It enables us to claim the invariant

$$\begin{array}{ll}
\text{(Lq7)} & i < \#sno.q \quad \wedge \quad loca.q(i)_3 \quad \wedge \quad n = sno.q(i) \\
& \Rightarrow \quad n < \#\mathtt{pm} \quad \wedge \quad \mathtt{pm}(n) = (q, i) \ ,
\end{array}$$

which clearly implies (Lq7a). Actually, the implication in (Lq7) can be replaced by an equivalence. The consequent implies the antecedent by (Kq6) and the additional invariant

$$\text{(Lq8)} \qquad n < \#\mathtt{pm} \quad \wedge \quad \mathtt{pm}(n) = (q, i) \quad \Rightarrow \quad loca.q(i)_3 \ .$$

Proof structure. Predicates (Lq0) and (Lq3) are inductive. Preservation of (Lq1) follows from (Jq3), (Lq0), (Lq2), (Lq3); preservation of (Lq2) follows from (Jq0), (Lq0); preservation of (Lq4) from (Lq0); preservation of (Lq5) from (Iq1), (Jq0), (Kq0), (Kq4), (Lq0), and (Lq4); preservation of (Lq6) from (Iq1), (Jq0), (Kq0), (Kq1), (Lq0), (Lq2), (Lq3), and (Lq4); preservation of (Lq7) from (Kq5) and (Lq0); preservation of (Lq8) from (Lq0) and (Lq4).

Summary. At this point, we have extended the implementation as specified in K_0 at the end of Section 5 to a specification $K_1 = (X_1, Y_1, N_1, P_1)$ by extending the state space with the auxiliary variables `time`, *lard*, *alt*, *sno*, $\mathtt{in}_4$, `hm`, `pm`, and *loca*. Here $\mathtt{in}_4$ stands for the auxiliary fourth components of the implementation array `in`. We have modified the next-state relation by replacing `Wr`, `Rd`, `MW`, `CU`, `Mrq`, and `MR` by by $\mathtt{Wr}_2$, $\mathtt{Rd}_2$, $\mathtt{MW}_2$, $\mathtt{CU}_1$, $\mathtt{Mrq}_1$, and $\mathtt{MR}_1$, respectively. The supplementary property P_1 is the straightforward translation of P_0. All steps of K_0 can also be done by K_1 and their effect on the variables of K_0 is unchanged. Therefore K_1 is an extension with history variables. Indeed, it is likely that K_1 is a history extension of K_0 in the sense of section 4.4. We did not verify this formally since we take one step further in the next subsection.

Specification K_1 satisfies the invariants (Iq*), (Jq*), (Kq*), and (Lq*), where the star serves as a wild card.

6.4 Inserting Serialization Points

In order to prove that the lazy caching algorithm implements sequential consistency as specified in Section 3, we still need an abstract memory `memA`, private variables *ser*, and serialization actions `WrD` and `RdD`, as in AuD.

Inspired by the invariants (Lq5) and (Lq6), we extend the state space with the auxiliary variables to instantiate n and i in (Lq5) and (Lq6):

var `cnt` : $Nat := 0$;
privar $ser : Nat := 0$.

Here `cnt` serves to count the total number of serialized write operations, while $ser.q$ holds the number of serialized operations of process q, and will play the role of *ser* of 3.4. We therefore introduce the serialization actions

`RdD` ▯ $ser < \#sno \ \wedge \ sno(ser) \leq$ `cnt` $\wedge \ \neg\, loca(ser)_3 \quad \rightarrow \quad ser$ `++` ;
`WrD` ▯ $ser < \#sno \ \wedge \ sno(ser) \leq$ `cnt` $\wedge \ loca(ser)_3 \quad \rightarrow$
ser `++` ; `cnt ++` .

Notice that these additional actions do not affect the implementation variables and project therefore to stutterings in the concrete behaviours.

In order to define a refinement mapping to Kab, we postulate the invariants

(Mq0) $ser.q \leq \#sno.q$,
(Mq1) `cnt` $\leq$ `time` .

If $loca.q(i)_3$ holds, $ser.q$ can only pass i by establishing $sno.q(i) <$ `cnt`. We therefore have the invariant

(Mq2) $i < ser.q \ \wedge \ loca.q(i)_3 \quad \Rightarrow \quad sno.q(i) <$ `cnt` .

Since we want that all read and write actions are serialized by means of `RdD` and `WrD` in the right order, we decide to keep the additional invariant

(Mq3) $ser.q < \#sno.q \quad \Rightarrow \quad$ `cnt` $\leq sno.q(ser.q)$.

In order to preserve (Mq3) under read actions, we decide also to keep the invariant

(Mq4) `cnt` $\leq lard.q$.

In order to preserve these invariants, we must strengthen the guard of `WrD` considerably. We first abbreviate the guard of `RdD` by

$$RdDg.q \quad \equiv \quad ser.q < \#sno.q \ \wedge \ sno.q(ser.q) \leq \texttt{cnt} \ \wedge \ \neg\, loca.q(ser.q)_3 \ .$$

For `WrD`, we aim at a condition that is independent of the acting process, and therefore we define the boolean state function

$SSL \quad = \quad$ `cnt` $< \#$`pm` $\wedge \ (\forall\, q \in Process : ssl.q)$, where
$ssl.q \quad = \quad \neg\, RdDg.q \ \wedge \ $ `cnt` $< lard.q$.

Suppose that *SSL* holds. We can write $\mathtt{pm}(\mathtt{cnt}) = (p, i)$. Then (Kq6) and (Lq8) imply that $i < \#sno.p$ and $sno.p(i) = \mathtt{cnt}$ and that $loca.p(i)_3$ holds. Now (Mq2) implies $ser.p \leq i$. Therefore, (Jq1) implies $sno.p(ser.p) \leq \mathtt{cnt}$. Using the definition of *SSL*, we get $sno.p(ser.p) = \mathtt{cnt}$ and $loca.p(ser.p)_3$. By (Lq7), this implies $ser.p = i$.

For every process $q \neq p$ with $ser.q < \#sno.q$, we have $\mathtt{cnt} \leq sno.q(ser.q)$. Moreover, $\mathtt{cnt} = sno.q(ser.q)$ would imply $loca.q(ser.q)$ and hence $p = \mathtt{pm}(\mathtt{cnt})_1 = q$ by (Lq7). It follows that $\mathtt{cnt} < sno.q(ser.q)$ for all $q \neq p$ with $ser.q < \#sno.q$. Also, $\mathtt{cnt} < lard.q$ for all q. This proves

$$\begin{array}{ll} \text{(WD0)} & SSL \ \wedge \ q = \mathtt{pm}(\mathtt{cnt})_1 \\ & \Rightarrow \ ser.q < \#sno.q \ \wedge \ sno.q(ser.q) = \mathtt{cnt} \ \wedge \ loca.q(ser.q)_3 \ , \\ \text{(WD1)} & SSL \ \wedge \ q \neq \mathtt{pm}(\mathtt{cnt})_1 \\ & \Rightarrow \ (ser.q < \#sno.q \ \Rightarrow \ \mathtt{cnt} < sno.q(ser.q)) \ \wedge \ \mathtt{cnt} < lard.q \ . \end{array}$$

It follows that execution of the body of `WrD` by process q would preserve (Mq3) and (Mq4). We therefore replace `WrD` by

$$\mathtt{WrD}_2 \qquad [\!] \ \ SSL \ \wedge \ self = \mathtt{pm}(\mathtt{cnt})_1 \ \ \rightarrow \ \ ser \ \mathtt{++} \, ; \ \ \mathtt{cnt} \ \mathtt{++} \, .$$

Since we want all actions to be serialized, we postulate that the commands `RdD` and `WrD` of all processes q are treated with weak fairness. We thus require

$$\text{(CF1)} \qquad \forall \, q \in Process : WF(\mathtt{RdD}.q) \wedge WF(\mathtt{WrD}.q) \ .$$

It turns out that the predicates (Mq0) up to (Mq4) are indeed invariants.

Proof structure. Predicate (WD0) is implied by (Jq1), (Kq6), (Lq7), (Lq8), (Mq2), and (Mq3); predicate (WD1) is implied by (Lq0), (Lq7), and (Mq3). Preservation of (Mq0) follows from (WD0); preservation of (Mq1) from (Kq5); preservation of (Mq2) from (WD0), (Lq0), and (Mq0); preservation of (Mq3) from (WD0), (WD1), (Jq1), (Lq0), (Lq1), (Mq0), (Mq1), and (Mq4); preservation of (Mq4) from (Iq3).

Summary. We have now extended specification K_1 as described at the end of Section 6.3 to a specification $K_2 = (X_2, Y_2, N_2, P_2)$ by extending the state space X_1 with the auxiliary variables `cnt` and *ser*. If N_1' is the extension of N_1 to the new state space that keeps the new variables unchanged, next-state relation N_2 is the union of N_1' with the union of the programs $\mathtt{RdD}.p$ and $\mathtt{WrD}_2.p$ for all processes p. The supplementary property P_2 is the conjunction of the translation of P_1 to the new state space with the additional fairness property (CF1). Specification K_2 has the invariants (Iq*), (Jq*), (Kq*), (Lq*), and (Mq*).

6.5 A Splitting Simulation

Let p_{20} be the projection function from X_2 to X_0 that retains the variables introduced in section 5, and forgets the values of all auxiliary variables: `time`, *lard*, *alt*, *sno*, $\mathtt{in}_4$, `hm`, `pm`, *loca*, *ser*, `cnt`. Let F_{02} be the converse relation that consists of the pairs (x, y) with $x = p_{20}(y)$.

Lemma 5 *Relation F_{02} is a splitting simulation $K_0 \rightarrowtriangle\!\!\!\!\rightarrow K_2$.*

Proof. For simplicity, we assume that the set of processes is finite. Verification of condition (F0) is straightforward. We now need to provide splittings $(i \in \mathbb{N} : A.i)$ and $(i \in \mathbb{N} : B.i)$ of K_0 and K_2, respectively. Since it has more alternatives, we start with K_2. We define the unfair alternative $B.0 \subseteq \mathit{step}(K_2)$ by

$$B.0 = (\bigcup q \in \mathit{Process} : \mathtt{MR}_1.q \cup \mathtt{CI}.q \cup \mathit{Env}.q) \ .$$

We define the other alternatives $B.i$ via some enumeration of the relations $\mathtt{Wr}_2.p$, $\mathtt{Rd}_2.p$, $\mathtt{MW}_2.p$, $\mathtt{CU}_1.p$, $\mathtt{Mrq}_1.p$, $\mathtt{WrD}_2.p$, $\mathtt{RdD}.p$, where p ranges over the (finite) set of the processes. The remaining sets $B.i$ are left empty. It is straightforward to verify that this defines a splitting of K_2.

We form the corresponding splitting $(i \in \mathbb{N} : A.i)$ of K_0, where the alternatives corresponding to $\mathtt{WrD}_2.p$ and $\mathtt{RdD}.p$ are left empty. Verification of condition (F1s) for the various alternatives is straightforward from the definitions. The alternatives $\mathtt{Wr}.p$, $\mathtt{Rd}.p$, $\mathtt{MW}.p$, $\mathtt{CU}.p$, $\mathtt{Mrq}.p$ are easily seen to satisfy condition (F2s) and hence (F2ns). Since they only modify the auxiliary variables $\mathtt{cnt}$ and *ser*, the alternatives $\mathtt{WrD}_2.p$ and $\mathtt{RdD}.p$ are stuttering alternatives and therefore also satisfy (F2ns). In conclusion, the proof is an administrative formality. □

7 Progress Expressed by "Leads-to-or"

In order to relate specification K_2 of section 6.4 to specification *Kab* of Section 3, we need to show that K_2 satisfies progress conditions analogous to the requirements (SF0), (AF1), and (AF2) of *Kab*. For this purpose, we use the concept of "leads-to" that was developed for UNITY in [CM88]. For the sake of (AF1), we need to generalize "leads-to" by allowing an escape route. All results claimed here have been verified mechanically with PVS.

Let a specification $K = (X, Y, N, P)$ be given. A predicate $Q \subseteq X$ is said to *lead to* a predicate R (notation $Q \ o\!\!\rightarrow R$) iff, in every behaviour, every state where $Q \wedge \neg R$ holds is eventually followed by a state where R holds. For a relation A, we define that Q *leads to* R *or* A (notation $Q \ o\!\!\rightarrow_A R$) to mean that, in every behaviour, every state where $Q \wedge \neg R$ holds is eventually followed by a state where R holds or by an A step.

We use the formal definitions

$$\begin{array}{lcl} (Q \ o\!\!\rightarrow R) & \equiv & \mathit{Beh}(K) \subseteq \Box\,(\llbracket\, Q \,\rrbracket \ \Rightarrow\ \Diamond \llbracket\, R \,\rrbracket) \ , \\ (Q \ o\!\!\rightarrow_A R) & \equiv & \mathit{Beh}(K) \subseteq \Box\,(\llbracket\, Q \,\rrbracket \ \Rightarrow\ \Diamond(\llbracket\, R \,\rrbracket \vee \llbracket\, A \,\rrbracket_2)) \ . \end{array}$$

It is easy to prove the following results:

Lemma 6 *(a)* $Q \ o\!\!\rightarrow R \ \equiv \ Q \ o\!\!\rightarrow_\emptyset R$.
(b) If $A \subseteq B$ *and* $Q \ o\!\!\rightarrow_A R$ *then* $Q \ o\!\!\rightarrow_B R$.
(c) Relation $o\!\!\rightarrow_A$ *is reflexive and transitive.*
(d) If $Q \subseteq R$ *then* $Q \ o\!\!\rightarrow R$.
(e) If J *is an invariant and* $Q \wedge J \ o\!\!\rightarrow R$. *Then* $Q \ o\!\!\rightarrow R$.
(f) $(\mathit{true} \ o\!\!\rightarrow_A R) \ \equiv \ \mathit{Beh}(K) \subseteq (\Box \Diamond \llbracket\, R \,\rrbracket) \cup (\Box \Diamond \llbracket\, A \,\rrbracket_2)$.

The next result shows that, in order to prove that a disjunction leads to something useful, it suffices to prove this for each of the disjuncts.

Lemma 7 *Let I be a set and let $(Q(i) : i \in I)$ be a family of predicates. Let R be a predicate with $Q(i)\ o{\rightarrow}_A\ R$ for all $i \in I$. Then $(\exists\, i : Q(i))\ o{\rightarrow}_A\ R$.*

The following facts can be easily deduced from the above ones.

Corollary 8 *(a) If $Q\ o{\rightarrow}_A\ S$ and $R\ o{\rightarrow}_A\ S$ then $(Q \vee R)\ o{\rightarrow}_A\ S$.*
(b) If $R\ o{\rightarrow}_A\ S$ and $Q \subseteq R \vee S$, then $Q\ o{\rightarrow}_A\ S$.
(c) If $Q\ o{\rightarrow}_A\ R \vee S$ and $S\ o{\rightarrow}_A\ R \vee T$, then $Q\ o{\rightarrow}_A\ R \vee T$.

Lemma 7 and Corollary (8)(c) together imply the following induction lemma:

Lemma 9 *Let R and $Q(k)$ be predicates for all $k \in \mathbb{N}$. Assume $Q(k{+}1)\ o{\rightarrow}_A\ R \vee Q(k)$ for all k. Then $(\exists\, k : Q(k))\ o{\rightarrow}_A\ R \vee Q(0)$.*

The simplest way to use weak fairness is contained in the following result.

Lemma 10 *Let relation B satisfy $Beh(K) \subseteq WF(B)$. Then we have*

$$true \quad o{\rightarrow} \quad \{y \mid \exists\, x : (x, y) \in B\} \vee disabled(B)\ .$$

Following [CM88], stability and, more generally, stable-unless relations are introduced to generalize this result. A predicate Q is called *stable* iff it keeps valid whenever it becomes valid in some behaviour. Predicate Q is called *stable unless R* iff, whenever Q is valid and R is not, Q remains valid or R becomes valid. Predicate Q is called *stable unless R or A* iff, whenever Q is valid and R is not, Q remains valid or R becomes valid or an A step is taken.

We use the following formal definitions, which are marginally weaker but much more practical. In these definitions, we let J range over the invariants of specification K.

$$\begin{array}{l} Q \text{ is stable} \quad \equiv \quad \exists\, J : Q \wedge J \subseteq wp(N, Q)\ , \\ Q \text{ stable-unless } R \quad \equiv \quad \exists\, J : Q \wedge J \wedge \neg R \subseteq wp(N, Q \vee R)\ , \\ Q \text{ stable-unless } R \text{ or } A \quad \equiv \quad \exists\, J : Q \wedge J \wedge \neg R \subseteq wp(N \setminus A, Q \vee R)\ . \end{array}$$

It is easy to see that Q is stable if and only if it is stable unless *false*.

The primary way to prove leads-to relations is by means of the next result:

Lemma 11 *Let predicates Q and R be given with Q stable unless R or A. Let relation B satisfy $Beh(K) \subseteq WF(B)$ and $Q \subseteq R \vee wp(B, R)$ and $Q \wedge disabled(B) \subseteq R$. Then $Q\ o{\rightarrow}_A\ R$.*

Proof. Assume that Q does not lead to R or A. Then there is a behaviour xs and a number n such that $xs(n) \in Q$ and $xs(k) \notin R$ and $(xs(k), xs(k+1)) \notin A$ for all $k \geq n$. Since Q is stable unless R or A, it follows by induction that $xs(k) \in Q$ for all $k \geq n$. Since $Q \wedge disabled(B)$ implies R, step B is continuously enabled beyond n. Therefore weak fairness of B implies that there is $k \geq n$ with $(xs(k), xs(k+1)) \in B$. Therefore $xs(k+1) \in R$, a contradiction.

Another application of *stable-unless* is in the finite conjunction lemma:

Lemma 12 *Let $R(i)$ be predicates for all $i \in I$, where I is a finite set. Let Q and T be predicates with $Q\ o{\rightarrow}_A\ T \vee R(i)$ and $R(i)$ stable unless T or A for all $i \in I$. Then $Q\ o{\rightarrow}_A\ T \vee (\forall\ i : R(i))$.*

The Progress-Safety-Progress Rule of [CM88] also generalizes to leads-to-or:

Lemma 13 *Let predicates Q, R, S, T be given with $Q\ o{\rightarrow}_A\ R$ and S stable unless T or A. Then $Q \wedge S\ o{\rightarrow}_A\ (R \wedge S) \vee T$.*

8 Construction of the Refinement Mapping

We now would like to construct a refinement mapping from specification K_2 of section 6.4 to specification *Kab* of Section 3.4. For this purpose, we need all invariants obtained in Section 6. In other words, we form the conjunction *Inv* of the invariants (Iq*), (Jq*), (Kq*), (Lq*), and (Mq*), and let $K_3 = res(K_2, Inv)$ be the corresponding invariant restriction, cf. section 4.3. Recall that the identity of *Inv* forms a strict simulation $\mathbf{1} : K_2 \rightarrow\!\triangleright\ K_3$. We are going to construct a refinement mapping $K_3 \rightarrow\!\triangleright\ Kab$.

The invariant (Mq1) enables us to define `memA` as the state function on K_3 by

$$\texttt{memA} \quad = \quad \texttt{hm}(\texttt{cnt})\ .$$

If we combine this function with the variables *ar*, *result*, *addr*, *val*, *loca*, *ser*, we obtain a function φ on the state space of K_3 with values in the state space of *Kab*. This function φ forgets the variables of the concrete system that do not occur in the abstract system. We claim that φ is a refinement mapping $K_3 \rightarrow\!\triangleright\ Kab$. The remainder of this section is devoted to the proof of this.

8.1 Safety for the Refinement Mapping

To prove that this function is indeed a refinement mapping, we need to verify three things. Firstly, the initial states of K_3 are mapped to initial states of *Kab*. This follows immediately from the construction.

Secondly, we need to prove that every step of K_3 is mapped into a, possibly stuttering, step of *Kab*. Since K_3 and *Kab* use the same environment programs *Env.p*, the steps of the environment clearly correspond. Using (Lq0) and (Mq0), it is easy to see that every $\texttt{Wr}_2$ step maps to a `Wr` step of *CoD*. The same holds for steps $\texttt{Rd}_2$. The steps $\texttt{MW}_2$, $\texttt{MR}_1$, $\texttt{Mrq}_1$, $\texttt{CU}_1$, and `CI` map to stuttering steps of *Kab*. The proof for `MW` uses the invariants (Kq0) and (Mq1). The other four cases are straightforward. The steps `WrD` in the two specifications correspond because of (Lq6) and (WD0). The steps `RdD` correspond because of (Lq5) and (Mq3). Mechanical verification of this essentially trivial step uncouvered a number of minor inconsistencies.

The third proof obligation is that every behaviour of specification of K_2 is mapped to a behaviour of specification *Kab*. This requires to prove that every concrete execution that satisfies the fairness properties (CF0) and (CF1) is mapped to a sequence of abstract states that satisfies the fairness properties (SF0), (AF1), and (AF2) of Section 3.

8.2 The System is Responsive

We begin with property (SF0) that $\Box\Diamond[\![\, addr.q = \bot \,]\!]$ for every process q: all behaviours satisfy always eventually $addr.q = \bot$, i.e. $true \; o\!\!\rightarrow addr.q = \bot$. This follows directly from weak fairness of $\mathtt{Wr}.q$ and Lemma 10 with $A := \mathtt{Wr}.q$.

The proof of property (AF1) is more complicated. It is based on weak fairness of $\mathtt{Mrq}.q$, $\mathtt{CU}.q$, $\mathtt{MW}.q$, and $\mathtt{Rd}.q$. We first use weak fairness of $\mathtt{Mrq}.q$ and Lemma 10 to obtain that always either $ar.q$ becomes $\bot$ or $\mathtt{Mrq}.q$ adds an item w with $w_1 = ar.q$ to $\mathtt{in}(q)$, as formalized in

$$\begin{array}{ll} (0) & \quad true \quad o\!\!\rightarrow \quad ar.q = \bot \;\vee\; (\exists\, k \in \mathbb{N} : Pin(q,k)) \text{ , where} \\ & \quad Pin(q,k) \quad \equiv \quad k < \#\mathtt{in}(q) \;\wedge\; \mathtt{in}(q)(k)_1 = ar.q \;. \end{array}$$

We now use Lemma 11 and weak fairness of $\mathtt{CU}.q$ to obtain that such an item moves forward in $\mathtt{in}(q)$, i.e., for every k,

$$Pin(q,k+1) \quad o\!\!\rightarrow \quad ar.q = \bot \;\vee\; Pin(q,k) \;.$$

By Lemma 9, this implies that

$$(1) \qquad (\exists\, k \in \mathbb{N} : Pin(q,k)) \quad o\!\!\rightarrow \quad ar.q = \bot \;\vee\; Pin(q,0) \;.$$

By Lemma 11, weak fairness of $\mathtt{CU}.q$ also implies that $Pin(q,0)$ leads to $ar.q = \bot \;\vee\; cache.q(ar.q) \neq \bot$. In combination with (0), (1), and Corollary 8, this gives

$$(2) \qquad true \quad o\!\!\rightarrow \quad ar.q = \bot \;\vee\; cache.q(ar.q) \neq \bot \;.$$

So, *true* leads to $ar.q = \bot$ or to the first two conjuncts of *inGuard*. In order to use weak fairness of $\mathtt{Rd}$, we need to cope with the other two conjuncts. We first use weak fairness of $\mathtt{MW}.q$ and Lemma 11 to prove that $out.q$ becomes shorter unless q itself writes, as formalized in

$$\begin{array}{l} Pout(q,k+1) \quad o\!\!\rightarrow_{\mathtt{Wr}.q} \quad Pout(q,k) \text{ , where} \\ Pout(q,k) \quad \equiv \quad \#out.q = k \;. \end{array}$$

By Lemma 9, this implies that $out.q$ becomes empty unless q itself writes:

$$(3) \qquad true \quad o\!\!\rightarrow_{\mathtt{Wr}.q} \quad out.q = \varepsilon \;.$$

Generalizing the last two conjuncts of *inGuard*, we introduce

$$Qin(q,k) \quad \equiv \quad out.q = \varepsilon \;\wedge\; (\forall\, i : k \leq i < \#\mathtt{in}(q) : \neg\, \mathtt{in}(q)(i)_3) \;.$$

By Lemma 11, weak fairness of $\mathtt{CU}.q$ implies that

$$Qin(q, k+1, n) \quad o{\rightarrow}_{\mathtt{Wr}.q} \quad Qin(q, k) \ .$$

By Lemma 9, this implies that $out.q = \varepsilon \; o{\rightarrow}_{\mathtt{Wr}.q} \; Qin(q, 0)$. By Corollary 8, this combines with (3) to yield

$$(4) \qquad true \quad o{\rightarrow}_{\mathtt{Wr}.q} \quad Qin(q, 0) \ .$$

We now apply Lemma 13 with $S : \mathtt{cache}(ar.q) \neq \bot$ and $T : ar.q = \bot$ to obtain

$$\begin{array}{l} \mathtt{cache}(ar.q) \neq \bot \quad o{\rightarrow}_{\mathtt{Wr}.q} \\ \quad (\mathtt{cache}(ar.q) \neq \bot \ \wedge \ Qin(q, 0)) \ \vee \ ar.q = \bot \ . \end{array}$$

In combination with (2) and the definition of *inGuard*, we thus obtain

$$(5) \qquad true \quad o{\rightarrow}_{\mathtt{Wr}.q} \quad inGuard.q \ \vee \ ar.q = \bot \ .$$

Finally, Lemma 11 together with weak fairness of $\mathtt{Rd}.q$ implies

$$inGuard.q \quad o{\rightarrow}_{\mathtt{Wr}.q} \quad ar.q = \bot \ .$$

By Corollary 8(c), this combines with formula (5) to yield

$$(6) \qquad true \quad o{\rightarrow}_{\mathtt{Wr}.q} \quad ar.q = \bot \ .$$

Therefore, by Lemma 6(f), all behaviours satisfy $(\Box\Diamond[\![\, ar.q = \bot \,]\!])\vee(\Box\Diamond[\![\, \mathtt{Wr}.q \,]\!]_2)$. This proves (AF1).

8.3 All Write Commands are Serialized

As a preparation for the proof of (AF2), we now aim at progress for serialization of the write commands. The write commands themselves are counted by `time`, whereas their serialization are counted by `cnt`. We thus aim at the formula

$$(7) \qquad \mathtt{time} \geq m \quad o{\rightarrow} \quad \mathtt{time} \geq m \ \wedge \ \mathtt{cnt} \geq m \ .$$

This is proved by means of the predicates

$$TiCn(m, j) \quad \equiv \quad \mathtt{time} \geq m \ \wedge \ \mathtt{cnt} \geq j \ .$$

Since the lefthand side of (7) equals $TiCn(m, 0)$ and the righthand side follows from $TiCn(m, m)$, it suffices to prove that $TiCn(m, j)$ leads to $TiCn(m, j+1)$ for all $j < m$.

For this purpose, we first claim that, for every process q,

$$(8) \qquad \mathtt{time} \geq m \quad o{\rightarrow} \quad lard.q \geq m \ .$$

To prove this, we introduce the predicates

$$\begin{array}{l} LaI(q, m, k) \quad \equiv \\ lard.q \geq m \ \vee \ (\exists\, i : i < k \ \wedge \ i < \#\mathtt{in}(q) \ \wedge \ \mathtt{in}(q)(i)_4 \geq m) \ . \end{array}$$

The lefthand predicate of (8) implies $(\exists\ k : LaI(q,m,k))$ because of the invariants (Iq4) and (Iq6). Clearly, $LaI(q,m,0)$ is equivalent to the righthand predicate of (8). The induction step is that $LaI(q,m,k+1)$ leads to $LaI(q,m,k)$, as follows from weak fairness of $\texttt{CU}$ for q, since $\texttt{CU}$ is enabled if and only if $\texttt{in}(q)$ is nonempty, and since $\texttt{CU}$ moves the elements of $\texttt{in}(q)$ to the front. Then Lemma 9 implies (8).

We subsequently use Lemma 13 and formula (8) to show that $TiCn(m,j)$ leads to $TiCn(m,j) \wedge lard.q \geq m$. This implies

$$(9) \qquad j < m \ \Rightarrow\ TiCn(m,j) \ \ o\!\!\rightarrow\ \ TiCn(m,j) \wedge lard.q > j\ .$$

We use weak fairness of $\texttt{RdD}(q)$ to prove that $ser.q \geq k$ leads to $\neg RdDg.q$ or $ser.q \geq k+1$. In order to show that $ser.q$ cannot grow indefinitely, we consider the predicates

$$\begin{array}{l} H(q,j) \ \equiv\ \texttt{cnt} \geq j\ \wedge\ lard.q > j\ , \\ H(q,j,n) \ \equiv\ H(q,j)\ \wedge\ (\forall\, i : n \leq i < \#sno.q\ \Rightarrow\ sno.q(i) > j)\ . \end{array}$$

These predicates are stable because of the invariants (Iq1) and (Iq3). We then use the PSP-rule to obtain

$$ser.q \geq k\ \wedge\ H(q,j,n)\ \ o\!\!\rightarrow\ \ (\neg RdDg.q\ \vee\ ser.q \geq k+1)\ \wedge\ H(q,j,n)\ .$$

Using Lemma 9, we get

$$H(q,j,n)\ \ o\!\!\rightarrow\ \ \texttt{cnt} \geq j+1\ \ \vee\ \ (\texttt{cnt} \geq j\ \wedge\ ssl.q)\ .$$

It is easy to see that $H(q,j) = (\exists\, n : H(q,j,n))$. Using Lemma 7, we therefore have

$$H(q,j)\ \ o\!\!\rightarrow\ \ \texttt{cnt} \geq j+1\ \ \vee\ \ (\texttt{cnt} \geq j\ \wedge\ ssl.q)\ .$$

Using formula (9) and transitivity, we obtain for $j < n$

$$TiCn(m,j)\ \ o\!\!\rightarrow\ \ \texttt{cnt} \geq j+1\ \ \vee\ \ (\texttt{cnt} \geq j\ \wedge\ ssl.q)\ .$$

Predicate $\texttt{cnt} \geq j \wedge ssl.q$ is stable unless $\texttt{cnt} \geq j+1$, because of (Lq0), (Iq1), and (Iq3), used at $\texttt{Rd}$, $\texttt{MW}$, and $\texttt{CU}$, respectively. Since the set of processes is finite and nonempty, we can use Lemma 12 to obtain for $j < n$

$$TiCn(m,j)\ \ o\!\!\rightarrow\ \ \texttt{cnt} \geq j+1\ \ \vee\ \ (\texttt{cnt} \geq j\ \wedge\ (\forall\, q : ssl.q))\ .$$

In view of the guard of command $\texttt{WrD}$, we define the predicate $C.p \equiv (\texttt{cnt} < \#\texttt{pm} \wedge p = \texttt{pm}(\texttt{cnt}))$. The conjunction $C.p\ \wedge\ \texttt{cnt} \geq j$ is stable unless $\texttt{cnt} \geq j+1$. Also, $\texttt{cnt} \geq j \wedge ssl.q$ is stable unless $\texttt{cnt} \geq j+1$. Therefore $C.p \wedge \texttt{cnt} \geq j \wedge ssl.q$ is stable unless $\texttt{cnt} \geq j+1$. Using weak fairness of $\texttt{WrD}(p)$, this implies

$$C.p\ \wedge\ \texttt{cnt} \geq j\ \wedge\ (\forall\, q : ssl.q)\ \ o\!\!\rightarrow\ \ \texttt{cnt} \geq j+1\ .$$

We now use Lemma 7 to obtain

$$(\exists\, p : C.p)\ \wedge\ \texttt{cnt} \geq j\ \wedge\ (\forall\, q : ssl.q)\ \ o\!\!\rightarrow\ \ \texttt{cnt} \geq j+1\ .$$

If $j < m$ then $\texttt{TiCn}(m,j) \subseteq TiCn(m,j+1) \vee (\exists\, p : C.p)$. Using stability and the PSP-rule, we thus get that $\texttt{TiCn}(m,j)$ leads to $\texttt{TiCn}(m,j+1)$ whenever $j < m$. This concludes the proof of formula (7).

8.4 Progress for Serialization

It remains to prove that the behaviours of specification K_2 map to sequences that satisfy (AF2), i.e., that $\#loca.q \geq n$ leads to $ser.q \geq n$. We first claim

$$(10) \qquad \#loca.q \geq n \quad o\!\!\rightarrow \quad \#sno.q \geq n \ .$$

This follows from Lemma 9 for the predicates

$$LoS(q,n,k) \quad \equiv \quad \#loca.q \geq n \ \wedge \ \#sno.q \geq n-k \ .$$

Indeed, $\#loca.q \geq n$ is equivalent to $(\exists\, k : LoS(q,n,k))$ and the righthand side of (10) follows from $LoS(q,n,0)$. Moreover, by (Lq0), the difference of $\#loca.q$ and $\#sno.q$ equals $\#out.q$. Since command `MW` of process q is enabled when $out.q$ is nonempty, and since `MW` decrements $\#out.q$ and increments $\#sno.q$, weak fairness of `MW` implies that $LoS(q,n,k+1)$ leads to $LoS(q,n,k)$.

We now aim at a second application of weak fairness for `RdD`. So, in view of the guard $RdDg$ of command `RdD`, we introduce the predicates

$$\begin{aligned} Ubs(q,n,m) \quad &\equiv \quad \#sno.q \geq n \\ &\quad \wedge (\forall\, i : i < n \ \Rightarrow \ sno.q(i) \leq m \ \wedge \ (loca.q(i)_3 \ \Rightarrow \ sno.q(i) < m)) \ . \end{aligned}$$

These predicates are stable since the lists $sno.q$ and $loca.q$ are only modified by adding elements, while $\#sno.q \leq \#loca.q$ because of (Lq0). On the other hand, using $m := \texttt{time}$ and the invariants (Jq0), (Lq2), and (Lq3), one can prove that

$$(\#sno.q \geq n) \ \subseteq \ (\exists\, m : \texttt{time} \geq m \ \wedge \ Ubs(q,n,m)) \ .$$

By formula (7) and Lemma 13, we have

$$\texttt{time} \geq m \ \wedge \ Ubs(q,n,m) \quad o\!\!\rightarrow \quad \texttt{cnt} \geq m \ \wedge \ Ubs(q,n,m) \ .$$

It follows from (Mq3) that we have

$$(ser.q < n \ \wedge \ \texttt{cnt} \geq m \ \wedge \ Ubs(q,n,m)) \ \subseteq \ RdDg.q \ .$$

Lemma 11 with weak fairness of $\texttt{RdD}.q$, together with Lemma 9, yields

$$\texttt{cnt} \geq m \ \wedge \ Ubs(q,n,m) \quad o\!\!\rightarrow \quad ser.q \geq n \ .$$

Putting all things together, using transitivity and Lemma 7, we finally obtain property (AF2) in the form

$$\#loca.q \geq n \quad o\!\!\rightarrow \quad ser.q \geq n \ .$$

8.5 Binding It All Together

Since we have now proved the properties (SF0) and (AF1) in Section 8.2 and (AF2) in Section 8.4, function φ maps every behaviour of K_3 to a behaviour of Kab. Therefore, φ is a refinement mapping $K_3 \rightarrowtriangle Kab$. So, the graph $\Gamma(\varphi)$ of φ is a strict simulation $K_3 \rightarrowtriangle Kab$. The composition $(\mathbf{1}; \Gamma(\varphi))$ is therefore a strict simulation $K_2 \rightarrowtriangle Kab$, and hence a simulation $K_2 \rightarrowtriangle\!\!\!\rightarrow Kab$.

In section 6.5, we formed a splitting simulation $F_{02} : K_0 \rightarrow\!\!\!\triangleright\!\!\!\triangleright K_2$, which is a simulation by Theorem 3. It follows that the composition $(F_{02}; \mathbf{1}; \Gamma(\varphi))$ is a simulation $K_0 \rightarrow\!\!\!\triangleright\!\!\!\triangleright Kab$. It is easy to verify that $(F_{02}; \mathbf{1}; \Gamma(\varphi)) \subseteq Fca$. Therefore, Fca is a simulation $K_0 \rightarrow\!\!\!\triangleright\!\!\!\triangleright Kab$. This concludes the proof of Theorem 4 in Section 5.

9 Aspects of the Mechanical Proof

The complete proof of Theorem 4 has been verified with the proof assistant PVS [OSRSC01]. The proof is available at `www.cs.rug.nl/~wim/mechver/lazyCaching`. The total proof hierarchy comprizes 412 lemmas. The proof can be split into three parts: the algorithm specific part, the theories about specifications and simulations, and some algorithm specific auxiliary theories.

The algorithm specific part took the greater effort. Its mechanical verification takes around 90% of the computing time. The core of the mechanical proof is the verification of the invariants claimed in Section 6. This is done in one PVS theory that contains 117 lemmas.

The second complicated effort is the verification of the progress properties of the algorithm in the Sections 8.2, 8.3, and 8.4. They are verified in a theory that holds 74 lemmas. The formalization of Section 5 and the verification of subsection 6.5 is done in one theory of 25 lemmas. Several small additional theories are used to finally prove Theorem 4 that Fca is a simulation $K_0 \rightarrow\!\!\!\triangleright\!\!\!\triangleright Kab$.

The general foundation is an extension of the PVS proofs to verify [Hes05a]. It was developed after most safety properties had been established, and in parallel with the verification of the progress properties and the construction of the simulation.

The relevant results of the Section 4, in particular the results on splitting simulations in 4.5, are verified in two theories of 28 lemmas. Actually, these theories are slightly stronger than needed in the present paper. They also justify the results of the companion paper [Hes06]. They are based on three theories about schedulers (48 lemmas), also presented in [Hes06]. The concepts developed in Section 2 and the lemmas developed in Section 7 are defined and verified in two theories that together hold 28 lemmas.

10 In Conclusion

The lazy caching (LC) problem of [Ger99] inspired us to develop splitting simulations in [Hes06], just as the serializable database interface (SDI) problem of [Sch92] inspired us to develop eternity extensions in [Hes02, Hes04, Hes05a].

The two problems LC and SDI are rather similar. Both are concerned with a number of client processes that interact concurrently with some kind of shared memory. Yet, whereas we needed eternity variables for SDI because of (our modelling of) the atomicity of SDI, it turns out that eternity variables are

not needed for LC. Similarly, the solution of LC in [LLOR99] uses stuttering variables rather than the prophecy variables of [AL91].

Inspired by [AL91], we obtained in [Hes05b] the following semantic completeness result: every simulation like $K_0 \twoheadrightarrow Kab$ can be factored over a clocking extension, an eternity extension, a stuttering history extension, and a refinement mapping. Semantic completeness indicates proving power, but should not be used to restrict the methodology. Indeed, in the present paper, the role of the composition of three first extensions has been taken over by a splitting simulation, which happens to be an extension but not a (stuttering) forward simulation in the sense of [Hes05b]. Indeed, we regard condition (F2) of section 4.4 as methodologically inadequate since it is phrased in terms of behaviours.

The importance of the proof assistant PVS for the project should not be underestimated. Indeed, handling the 33 invariants of Section 6 and the 26 leads-to relations of section 8 would hardly be feasible without mechanical support. Also, the proofs in [Hes06] of the two theorems of Section 4 are delicate enough to justify the use of PVS. The results of Section 7 are relatively harmless and we did them with PVS just for completeness. In mathematics much more complicated theorems and theories have been proved without such safety belts, but when an adequate proof assistant is available, it is advisable to use it.

Acknowledgements.
Constructive criticisms of four anonymous referees are gratefully acknowledged.

References

[ABM93] Y. Afek, G. Brown, and M. Merrit. Lazy caching. *ACM Trans. Program. Lang. Syst.*, 15:182–206, 1993.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82:253–284, 1991.

[Aro01] T. Arons. Using timestamping and history variables to verify sequential consistency. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris*, volume 2102 of *LNCS*, pages 223–235, New York, 2001. Springer.

[Bri99] E. Brinksma. Cache consistency by design. *Distr. Comput.*, 12:61–74, 1999.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison–Wesley, 1988.

[Ger99] R. Gerth. Sequential consistency and the lazy caching algorithm. *Distr. Comput.*, 12:57–59, 1999.

[Gra99] S. Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Distr. Comput.*, 12:75–90, 1999.

[Hes02] W.H. Hesselink. Eternity variables to simulate specifications. In E.A. Boiten and B. Moeller, editors, *MPC 2002*, volume 2386 of *LNCS*, pages 117–130, New York, 2002. Springer.

[Hes04] W.H. Hesselink. Using eternity variables to specify and prove a serializable database interface. *Sci. Comput. Program.*, 51:47–85, 2004.

[Hes05a] W.H. Hesselink. Eternity variables to prove simulation of specifications. *ACM Trans. on Comp. Logic*, 6:175–201, 2005.

[Hes05b] W.H. Hesselink. Universal extensions to simulate specifications. In preparation, see `www.cs.rug.nl/~wim/pub/mans.html`, 2005.

[Hes06] W.H. Hesselink. Splitting forward simulations to cope with liveness. *Acta Inf.*, 2006. (to appear).

[HHS86] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *ESOP 86*, volume 213 of *LNCS*, pages 187–196, New York, 1986. Springer.

[JPR99] B. Jonsson, A. Pnueli, and C. Rump. Proving refinement using transduction. *Distr. Comput.*, 12:129–149, 1999.

[JPZ99] W. Janssen, M. Poel, and J. Zwiers. The compositional approach to sequential consistency and lazy caching. *Distr. Comput.*, 12:105–127, 1999.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32:32–45, 1989.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, 1994.

[LD99] G. Lowe and J. Davies. Using CSP to verify sequential consistency. *Distr. Comput.*, 12:91–103, 1999.

[LLOR99] P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching in TLA. *Distr. Comput.*, 12:151–174, 1999.

[LV95] N. Lynch and F. Vaandrager. Forward and backward simulations, part I: untimed systems. *Inf. Comput.*, 121:214–233, 1995.

[Mer99] M. Meritt. Introduction. *Distr. Comput.*, 12:55–56, 1999.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Int. Joint Conf. on Artificial Intelligence*, pages 481–489. British Comp. Soc., 1971.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer, New York, 1992.

[OSRSC01] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference*, 2001. `http://pvs.csl.sri.com`

[Sch92] F. B. Schneider. Introduction. *Distr. Comput.*, 6:1–3, 1992.