A Formal Reduction for Lock-Free Parallel Algorithms

Hui Gao and Wim H. Hesselink

Department of Mathematics and Computing Science, University of Groningen
P.O. Box 800, 9700 AV Groningen, The Netherlands
{hui,wim}@cs.rug.nl

Abstract. On shared memory multiprocessors, synchronization often turns out to be a performance bottleneck and the source of poor fault-tolerance. Lock-free algorithms can do without locking mechanisms, and are therefore desirable. Lock-free algorithms are hard to design correctly, however, even when apparently straightforward. We formalize Herlihy's methodology [13] for transferring a sequential implementation of any data structure into a lock-free synchronization by means of synchronization primitives *Load-linked* (*LL*)/*store-conditional* (*SC*). This is done by means of a reduction theorem that enables us to reason about the general lock-free algorithm to be designed on a higher level than the synchronization primitives. The reduction theorem is based on refinement mapping as described by Lamport [10] and has been verified with the higher-order interactive theorem prover PVS. Using the reduction theorem, fewer invariants are required and some invariants are easier to discover and easier to formulate.

The lock-free implementation works quite well for small objects. However, for large objects, the approach is not very attractive as the burden of copying the data can be very heavy. We propose two enhanced lock-free algorithms for large objects in which slower processes don't need to copy the entire object again if their attempts fail. This results in lower copying overhead than in Herlihy's proposal.

Keywords & Phrases: Distributed algorithms, Lock-free, Simulation, Refinement mapping

1 Introduction

On shared-memory multiprocessors, processes coordinate with each other via shared data structures. To ensure the consistency of these concurrent objects, processes need a mechanism for synchronizing their access. In such a system the programmer typically has to explicitly synchronize access to shared data by different processes to ensure correct behaviors of the overall system, using synchronization primitives such as semaphores, monitors, guarded statements, mutex locks, etc. Consequently the operations of different processes on a shared data structure should appear to be serialized: if two operations execute simultaneously, the system guarantees the same result as if one of them is arbitrarily executed before the other.

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 44–56, 2004.

Due to blocking, the classical synchronization paradigms using locks can incur many problems such as convoying, priority inversion and deadlock. A *lock-free* (also called non-blocking) implementation of a shared object guarantees that within a finite number of steps always some process trying to perform an operation on the object will complete its task, independently of the activity and speed of other processes [13]. As lock-free synchronizations are built without locks, they are immune from the aforementioned problems. In addition, lock-free synchronizations can offer progress guarantees. A number of researchers [1, 4, 5, 13–15] have proposed techniques for designing lock-free implementations. The basis of these techniques is using some synchronization primitives such as *compare-and-swap* (CAS), or *Load-linked* (LL)/*store-conditional* (SC).

Typically, the implementation of the synchronization operations is left to the designer, who has to decide how much of the functionality to implement in software using system libraries. The high-level specification gives lots of freedom about how a result is obtained. It is constructed in some mechanical way that guarantees its correctness and then the required conditions are automatically satisfied [3]. We reason about a high-level specification of a system, with a large grain of atomicity, and hope to deduce an implementation, a low-level specification, which must be fine grained enough to be translated into a computer program that has all important properties of the high-level specification.

However, the correctness properties of an implementation are seldom easy to verify. Our previous work [6] shows that a proof may require unreasonable amounts of effort, time, or skill. We therefore develop a reduction theorem that enables us to reason about a lock-free program to be designed on a higher level than the synchronization primitives. The reduction theorem is based on refinement mappings as described by Lamport [10], which are used to prove that a lower-level specification correctly implements a higher-level one. Using the reduction theorem, fewer invariants are required and some invariants are easier to discover and easier to formulate, without considering the internal structure of the final implementation. In particular, nested loops in the algorithm may be treated as one loop at a time.

2 Lock-Free Transformation

The machine architecture that we have in mind is based on modern shared-memory multiprocessors that can access a common shared address space. There can be several processes running on a single processor. Let us assume there are P (≥ 1) concurrently executing sequential processes.

Synchronization primitives LL and SC, proposed by Jensen et al. [2], have found widespread acceptance in modern processor architectures (e.g. MIPS II, PowerPC and Alpha architectures). They are a pair of instructions, closely related to the CAS, and together implement an atomic Read/Write cycle. Instruction LL first reads a memory location, say X, and marks it as "reserved" (not "locked"). If no other processor changes the contents of X in between, the subsequent SC operation of the same processor succeeds and modifies the value stored;

otherwise it fails. There is also a validate instruction VL, used to check whether X was not modified since the corresponding LL instruction was executed. Implementing VL should be straightforward in an architecture that already supports SC. Note that the implementation does not access or manipulate X other than by means of $LL/SC/VL$. Moir [12] showed that $LL/SC/VL$ can be constructed on any system that supports either LL/SC or CAS. A shared variable X only accessed by $LL/SC/VL$ operations can be regarded as a variable that has an associated shared set of process identifiers $V.X$, which is initially empty. The semantics of LL, VL and SC are given by equivalent atomic statements below.

```
proc $LL$(ref $X$ : val) : val =
    ⟨  $V.X := V.X \cup \{self\}$; return $X$; ⟩

proc $VL$(ref $X$ : val) : boolean =
    ⟨  return $(self \in V.X)$ ⟩

proc $SC$(ref $X$ : val; in $Y$ : val) : boolean =
    ⟨  if $self \in V.X$ then $V.X := \emptyset$; $X := Y$; return $true$
       else return $false$; fi ⟩
```

where *self* is the process identifier of the acting process.

At the cost of copying an object's data before an operation, Herlihy [13] introduced a general methodology to transfer a sequential implementation of any data structure into a lock-free synchronization by means of synchronization primitives LL and SC. A process that needs access to a shared object pointed by X performs a loop of the following steps:(1) read X using an LL operation to gain access to the object's data area; (2) make a private copy of the indicated version of the object (this action need not be atomic); (3) perform the desired operation on the private copy to make a new version; (4) finally, call a SC operation on X to attempt to swing the pointer from the old version to the new version. The SC operation will fail when some other process has modified X since the LL operation, in which case the process has to repeat these steps until consistency is satisfied. The algorithm is non-blocking because at least one out of every P attempts must succeed within finite time. Of course, a process might always lose to some faster process, but this is often unlikely in practice.

3 Reduction

We assume a universal set $\mathcal{V}$ of typed variables, which is called the *vocabulary*. A state s is a type-consistent interpretation of $\mathcal{V}$, mapping variables $v \in \mathcal{V}$ to values $s[\![v]\!]$. We denote by Σ the set of all states. If $\mathcal{C}$ is a command, we denote by $\mathcal{C}_p$ the transition $\mathcal{C}$ executed by process p, and $s[\![\mathcal{C}_p]\!]t$ indicates that in state s process p can do a step $\mathcal{C}$ that establishes state t. When discussing the effect of a transition $\mathcal{C}_p$ from state s to state t on a variable v, we abbreviate $s[\![v]\!]$ to v and $t[\![v]\!]$ to v'. We use the abbreviation $Pres(V)$ for $\bigwedge_{v \in V}(v' = v)$ to denote that all variables in the set V are preserved by the transition. Every private variable name can be extended with the suffix "." + "*process identifier*". We sometimes use indentation to eliminate parentheses.

3.1 Observed Specification

In practice, the specification of systems is concerned rather with externally visible behavior than computational feasibility. We assume that all levels of specifications under consideration have the same observable state space Σ_0, and are interpreted by their observation functions $\Pi : \Sigma \rightarrow \Sigma_0$. Every specification can be modeled as a five-tuple $(\Sigma, \Pi, \Theta, \mathcal{N}, \mathcal{L})$ where $(\Sigma, \Theta, \mathcal{N})$ is the *transition system* [16] and $\mathcal{L}$ is the supplementary property of the system (i.e., a predicate on Σ^{ω}).

The supplementary constraint $\mathcal{L}$ is imposed since the transition system only specifies safety requirements and has no kind of fairness conditions or liveness assumptions built into it. Since, in reality, a stuttering step might actually perform modifications to some internal variables in internal states, we do allow stuttering transitions (where the state does not change) and the next-state relation is therefore reflexive. A finite or infinite sequence of states is defined to be an *execution* of system $(\Sigma, \Pi, \Theta, \mathcal{N}, \mathcal{L})$ if it satisfies initial predicate Θ and the next-state relation $\mathcal{N}$ but not necessarily the requirements of the supplementary property $\mathcal{L}$. We define a *behavior* to be an infinite execution that satisfies the supplementary property $\mathcal{L}$. A (concrete) specification $\mathcal{S}_c$ *implements* a (abstract) specification S_a iff every externally visible behavior allowed by $\mathcal{S}_c$ is also allowed by S_a. We write $Beh(\mathcal{S})$ to denote the set of behaviors of system $\mathcal{S}$.

3.2 Refinement Mappings

A *refinement mapping* from a lower-level specification $\mathcal{S}_c = (\Sigma_c, \Pi_c, \Theta_c, \mathcal{N}_c, \mathcal{L}_c)$ to a higher-level specification $\mathcal{S}_a = (\Sigma_a, \Pi_a, \Theta_a, \mathcal{N}_a, \mathcal{L}_a)$, written $\phi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$, is a mapping $\phi : \Sigma_c \rightarrow \Sigma_a$ that satisfies:

1. ϕ preserves the externally visible state component: $\Pi_a \circ \phi = \Pi_c$.
2. ϕ is a *simulation*, denoted $\phi : \mathcal{S}_c \preccurlyeq \mathcal{S}_a$:
   ① ϕ takes initial states into initial states: $\Theta_c \Rightarrow \Theta_a \circ \phi$.
   ② $\mathcal{N}_c$ is mapped by ϕ into a transition (possibly stuttering) allowed by $\mathcal{N}_a$: $\mathcal{Q} \wedge \mathcal{N}_c \;\Rightarrow\; \mathcal{N}_a \circ \phi$, where $\mathcal{Q}$ is an invariant of $\mathcal{S}_c$.
3. ϕ maps behaviors allowed by $\mathcal{S}_c$ into behaviors that satisfy $\mathcal{S}_a$'s supplementary property: $\forall\ \sigma \in Beh(\mathcal{S}_c) : \mathcal{L}_a(\phi(\sigma))$.

Below we need to exploit the fact that the simulation only quantifies over all reachable states of the lower-level system, not all states. We therefore explicitly allow an invariant $\mathcal{Q}$ in condition 2 ②. The following theorem is stated in [11].

Theorem 1. *If there exists a refinement mapping from $\mathcal{S}_c$ to $\mathcal{S}_a$, then $\mathcal{S}_c$ implements $\mathcal{S}_a$.*

Refinement mappings give us the ability to reduce an implementation by reducing its components in relative isolation, and then gluing the *reductions* together with the same structure as the implementation. Atomicity guarantees that a parallel execution of a program gives the same results as a sequential and non-deterministic execution. This allows us to use the refinement calculus for stepwise refinement of transition systems [8]. Essentially, the reduction theorem allows us to design and verify the program on a higher level of abstraction. The big advantage is that substantial pieces of the concrete program can be dealt with as atomic statements on the higher level.

The refinement relation is transitive, which means that we don't have to reduce the implementation in one step, but can proceed from the implementation to the specification through a series of smaller steps.

3.3 Correctness

The safety properties satisfied by the program are completely determined by the initial predicate and the next-state relation. This is described by Theorem 2, which can be easily verified.

Theorem 2. *Let $\mathcal{P}_c$ and $\mathcal{P}_a$ be safety properties for $\mathcal{S}_c$ and $\mathcal{S}_a$ respectively. The verification of a concrete judgment $(\Sigma_c, \Theta_c, \mathcal{N}_c) \models \mathcal{P}_c$ can be reduced to the verification of an abstract judgment $(\Sigma_a, \Theta_a, \mathcal{N}_a) \models \mathcal{P}_a$, if we can exhibit a simulation ϕ mapping from Σ_c to Σ_a that satisfies $\mathcal{P}_a \circ \phi \Rightarrow \mathcal{P}_c$.*

We make a distinction between safety and liveness properties (See [10] for the proof schemes). The proof of liveness relies on the fairness conditions associated with a specification. The purpose for fairness conditions is to rule out executions where the system idles indefinitely with control at some internal point of a procedure and with some transition of that procedure enabled. Fairness arguments usually depend on safety properties of the system.

4 A Lock-Free Pattern

We propose a pattern that can be universally employed for a lock-free construction in order to synchronize access to a shared node of *nodeType*. The interface $\mathcal{S}_a$ is shown in Fig. 1, where the following statements are taken as a schematic representation of segments of code:

1. *noncrit*(`ref` *pub* : *aType*, *priv* : *bType*; `in` *tm* : *cType*; `out` x : $1..N$) : representing an atomic non-critical activity on variables *pub* and *priv* according to the value of *tm*, and choosing an index x of a shared node to be accessed.
2. *guard*(`in` X : *nodeType*, *priv* : *bType*) a non-atomic boolean test on the variable X of *nodeType*. It may depend on private variable *priv*.
3. *com*(`ref` X : *nodeType*; `in` *priv* : *bType*; `out` *tm* : *cType*) : a non-atomic action on the variable X of *nodeType* and private variable *tm*. It is allowed to inspect private variable *priv*.

```
CONSTANT
  P = number of processes; N = number of nodes
Shared Variables:
  pub: aType; Node: array [1..N] of nodeType;
Private Variables:
  priv: bType; pc: $\{a1,\ a2\}$; x: 1..N; tm: cType;
Program:
      loop
  $a1$:   noncrit(pub, priv, tm, x);
  $a2$:   $\langle$ if guard(Node[x], priv) then com(Node[x], priv, tm); fi $\rangle$
      end
Initial conditions $\Theta_a$ : $\forall$ p:1..P: pc = a1
Liveness  $\mathcal{L}_a$ : $\Box$ ( $pc = a2 \longrightarrow \Diamond\ pc = a1$ )
```

Fig. 1. Interface $\mathcal{S}_a$

```
CONSTANT
  P = number of processes; N = number of nodes
Shared Variables:
  pub: aType; node: array [1..N+P] of nodeType;
  indir: array [1..N] of 1..N+P;
Private Variables:
  priv: bType; pc: $[c1..\ c7]$;
  x: 1..N; mp, m: 1..N+P; tm, tm1: cType;
Program:
      loop
  $c1$:   noncrit(pub, priv, tm, x);
        loop
  $c2$:     m := LL(indir[x]);
  $c3$:     read(node[mp], node[m]);
  $c4$:     if guard(node[mp], priv) then
  $c5$:        com(node[mp], priv, tm1);
  $c6$:        if SC(indir[x], mp) then
                 mp := m; tm := tm1; break;
              fi
  $c7$:     else
              if VL(indir[x]) then break; fi
           fi
        end
      end
Initial conditions $\Theta_c$ :
  ($\forall$ p:1..P: pc = c1 $\wedge$ mp=N+p) $\wedge$ ($\forall$ i:1..N: indir[i]=i)
Liveness  $\mathcal{L}_c$ : $\Box$ ( $pc = c2 \longrightarrow \Diamond\ pc = c1$ )
```

Fig. 2. Lock-free implementation $\mathcal{S}_c$ of $\mathcal{S}_a$

The action enclosed by angular brackets $\langle \ldots \rangle$ is defined as atomic. The private variable x is intended only to determine the node under consideration, the private variable tm is intended to hold the result of the critical computation com, if executed. By means of Herlihy's methodology, we give a lock-free implementation $\mathcal{S}_c$ of interface $\mathcal{S}_a$ in Fig. 2. In the implementation, we use some other schematic representations of segments of code, which are described as follows:

4. $read(\texttt{ref}\ X : nodeType,\ \texttt{in}\ Y : nodeType)$: a non-atomic read operation that reads the value from the variable Y of $nodeType$ to the variable X of $nodeType$, and does nothing else. If Y is modified during $read$, the resulting value of X is unspecified but type correct, and no error occurs.
5. LL, SC and VL : atomic actions as we defined before.

Typically, we are not interested in the internal details of these schematic commands but in their behavior with respect to lock-freedom. In $\mathcal{S}_c$, we declare P extra shared nodes for private use (one for each process). Array $indir$ acts as pointers to shared nodes. $node[mp.p]$ can always be taken as a "private" node (other processes can read but not modify the content of the node) of process p though it is declared publicly. If some other process successfully updates a shared node while an active process p is copying the shared node to its "private" node, process p will restart the inner loop, since its private view of the node is not consistent anymore. After the assignment $mp := m$ at line $c6$, the "private" node becomes shared and the node shared previously (which contains the old version) becomes "private".

Formally, we introduce N_c as the relation corresponding to command $noncrit$ on $(aType \times bType \times cType,\ aType \times bType \times 1..N)$, P_g as the predicate computed by $guard$ on $nodeType \times bType$, R_c as the relation corresponding to com on $(nodeType \times bType, nodeType \times cType)$, and define

$$\begin{aligned}
&\Sigma_a \triangleq (Node[1..N],\ pub) \times (pc,\ x,\ priv,\ tm)^P,\\
&\Sigma_c \triangleq (node[1..N{+}P],\ indir[1..N],\ pub) \times (pc,\ x,\ mp,\ m,\ priv,\ tm,\ tm1)^P,\\
&\Pi_a(\Sigma_a) \triangleq (Node[1..N],\ pub), \quad \Pi_c(\Sigma_c) \triangleq (node[indir[1..N]],\ pub),\\
&\mathcal{N}_a \triangleq \textstyle\bigvee_{0\le i\le 2} \mathcal{N}_{ai}\ , \quad \mathcal{N}_c \triangleq \textstyle\bigvee_{1\le i\le 7} \mathcal{N}_{ci}\ ,
\end{aligned}$$

The transitions of the abstract system can be described: $\forall s, t : \Sigma_a,\ p : 1..P$:

$$\begin{aligned}
s[\![(\mathcal{N}_{a0})_p]\!]t \triangleq\ & s = t \qquad \text{(to allow stuttering)}\\
s[\![(\mathcal{N}_{a1})_p]\!]t \triangleq\ & pc.p = a1\ \wedge pc'.p = a2\ \wedge Pres(\mathcal{V} - \{pub,\ priv.p,\ pc.p,\ x.p\})\\
& \wedge\ ((pub,\ priv.p,\ tm.p), (pub,\ priv.p,\ x.p)') \in N_c\\
s[\![(\mathcal{N}_{a2})_p]\!]t \triangleq\ & pc.p = a2\ \wedge\ pc'.p = a1\ \wedge\ (P_g(Node[x],\ priv.p)\\
& \quad \wedge\ ((Node[x],\ priv.p), (Node[x],\ tm.p)') \in R_c\\
& \quad \wedge\ Pres(\mathcal{V} - \{pc.p,\ Node[x],\ tm.p\})\\
& \vee \neg P_g(Node[x],\ priv.p)\ \wedge\ Pres(\mathcal{V} - \{pc.p\})).
\end{aligned}$$

The transitions of the concrete system can be described in the same way. Here we only provide the description of the step that starts in $c6$: $\forall s, t : \Sigma_c,\ p : 1..P$:

$$\begin{aligned}
s[\![(\mathcal{N}_{c6})_p]\!]t \triangleq\ & pc.p = c6\ \wedge\ (p \in V.indir[x.p]\\
& \quad \wedge\ pc'.p = c1\ \wedge\ (indir[x.p])' = mp.p\ \wedge\ mp'.p = m.p\\
& \quad \wedge\ tm'.p = tm1.p\ \wedge\ (V.indir[x.p])' = \emptyset\\
& \quad \wedge\ Pres(\mathcal{V} - \{pc.p,\ indir[x.p],\ mp.p,\ tm.p,\ V.indir[x.p]\})\\
& \vee\ \ p \notin V.indir[x.p]\ \wedge\ pc'.p = c2\ \wedge\ Pres(\mathcal{V} - \{pc.p\}))
\end{aligned}$$

4.1 Simulation

According to Theorem 2, the verification of a safety property of concrete system $\mathcal{S}_c$ can be reduced to the verification of the corresponding safety property of abstract system $\mathcal{S}_a$ if we can exhibit the existence of a simulation between them.

Theorem 3. *The concrete system $\mathcal{S}_c$ defined in Fig. 2 is simulated by the abstract system $\mathcal{S}_a$ defined in Fig. 1, that is, $\exists \phi : \mathcal{S}_c \preccurlyeq \mathcal{S}_a$.*

Proof: We prove Theorem 3 by providing a simulation. The simulation function ϕ is defined by showing how each component of the abstract state (i.e. state of Σ_a) is generated from components in the concrete state (i.e. state of Σ_c). We define ϕ : the concrete location $c1$ is mapped to the abstract location $a1$, while all other concrete locations are mapped to $a2$; the concrete shared variable $node[indir[x]]$ is mapped to the abstract shared variable $Node[x]$, and the remaining variables are all mapped to the identity of the variables occurring in the abstract system.

The assertion that the initial condition Θ_c of the concrete system implies the initial condition Θ_a of the abstract system follows easily from the definitions of Θ_c, Θ_a and ϕ.

The central step in the proof of simulation is to prove that every atomic step of the concrete system simulates an atomic step of the abstract system. We therefore need to associate each transition in the concrete system with the transition in the abstract system.

It is easy to see that the concrete transition $\mathcal{N}_{c1}$ simulates $\mathcal{N}_{a1}$ and that $\mathcal{N}_{c2}$, $\mathcal{N}_{c3}$, $\mathcal{N}_{c4}$, $\mathcal{N}_{c5}$, $\mathcal{N}_{c6}$ with precondition "$self \notin V.indir[x.self]$", and $\mathcal{N}_{c7}$ with precondition "$self \notin V.indir[x.self]$" simulate a stuttering step $\mathcal{N}_{a0}$ in the abstract system. E.g., we prove that $\mathcal{N}_{c6}$ executed by any process p with precondition "$p \notin V.indir[x.p]$" simulates a stuttering step in the abstract system. By the mechanism of SC, an active process p will only modify its program counter $pc.p$ from $c6$ to $c2$ when executing $\mathcal{N}_{c6}$ with precondition "$p \notin V.indir[x.p]$". According to the mapping of ϕ, both concrete locations $c6$ and $c2$ are mapped to abstract location $a2$. Since the mappings of the pre-state and the post-state to the abstract system are identical, $\mathcal{N}_{c6}$ executed by process p with precondition "$p \notin V.indir[x.p]$" simulates the stuttering step $\mathcal{N}_{a0}$ in the abstract system.

The proof for the simulations of the remaining concrete transitions is less obvious. Since simulation applies only to transitions taken from a reachable state, we postulate the following invariants in the concrete system $\mathcal{S}_c$:

Q1: $(p \neq q \Rightarrow mp.p \neq mp.q) \;\wedge\; (indir[y] \neq mp.p)$
$\quad\wedge\; (y \neq z \Rightarrow indir[y] \neq indir[z])$

Q2: $pc.p = c6 \;\wedge\; p \in V.indir[x.p]$
$\quad\Rightarrow\; ((node[m.p],\ priv.p),\ (node[mp.p],\ tm1.p)) \in R_c$

Q3: $pc.p = c7 \;\wedge\; p \in V.indir[x.p] \;\Rightarrow\; \neg\, P_g(node[m.p],\ priv.p)$

Q4: $pc.p \in [c3..c7] \;\wedge\; p \in V.indir[x.p] \;\Rightarrow\; m.p = indir[x.p]$

Q5: $pc.p \in \{c4,\ c5\} \;\wedge\; p \in V.indir[x.p] \;\Rightarrow\; node[m.p] = node[mp.p]$

Q6: $pc.p = \{c5,\ c6\} \;\Rightarrow\; P_g(node[mp.p],\ priv.p)$

In the invariants, the free variables p and q range over $1..P$, and the free variables y and z range over $1..N$. Invariant $Q1$ implies that, for any process q, $node[mp.q]$ can be indeed treated as a "private" node of process q since only process q can modify that. Invariant $Q4$ reflect the mechanism of the synchronization primitives LL and SC.

With the help of those invariants above, we have proved that $\mathcal{N}_{c6}$ and $\mathcal{N}_{c7}$ executed by process p with precondition "$p \in V.indir[x.p]$" simulate the abstract step $\mathcal{N}_{a2}$ in the abstract system. For reasons of space we refer the interested reader to [7] for the complete mechanical proof. □

4.2 Refinement

Recall that not all simulation relations are refinement mappings. According to the formalism of the reduction, it is easy to verify that ϕ preserves the externally visible state component. For the refinement relation we also need to prove that the simulation ϕ maps behaviors allowed by S_c into behaviors that satisfy S_a's liveness property, that is, $\forall \sigma \in Beh(\mathcal{S}_c) : \mathcal{L}_a(\phi(\sigma))$. Since ϕ is a simulation, we deduce

$$\begin{array}{rcl} \sigma \models \mathcal{L}_c & \equiv & \sigma \models \Box(pc = c2 \longrightarrow \Diamond pc = c1) \\ & \Rightarrow & \sigma \models \Box(pc \in [c2..c7] \longrightarrow \Diamond pc = c1) \\ & \Rightarrow & \phi(\sigma) \models \Box(pc = a2 \longrightarrow \Diamond pc = a1) \\ & \equiv & \mathcal{L}_a(\phi(\sigma)) \end{array}$$

Consequently, we have our main reduction theorem:

Theorem 4. *The abstract system $\mathcal{S}_a$ defined in Fig. 1 is refined by the concrete system $\mathcal{S}_c$ defined in Fig. 2, that is, $\exists \phi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$.*

The liveness property $\mathcal{L}_c$ of concrete system $\mathcal{S}_c$ can also be proved under the assumption of the strong fairness conditions and the following assumption:

$$\begin{array}{l} \Box\ (\Box pc.p \in [c2..c7] \wedge \Box\Diamond p \in V.indir[x.p] \\ \qquad \longrightarrow\ \Diamond(pc.p = c6 \vee pc.p = c7) \wedge p \in V.indir[x.p]). \end{array}$$

The additional assumption indicates that for every process p, when process p remains in the loop from $c2$ to $c7$ and executes $c2$ infinitely often, it will eventually succeed in reaching $c6$ or $c7$ with precondition "$p \in V.indir[x.p]$".

5 Large Object

To reduce the overhead of failing non-blocking operations, Herlihy [13] proposes an exponential back-off policy to reduce useless parallelism, which is caused by failing attempts. A fundamental problem with Herlihy's methodology is the overhead that results from making complete copies of the entire object ($c3$ in Fig. 2) even if only a small part of an object has been changed. For a large object this may be excessive.

We therefore propose two alternatives given in Fig. 3. For both algorithms the fields of the object are divided into W disjoint logical groups such that if one field is modified then other fields in the same group may be modified simultaneously. We introduce an additional field *ver* in *nodeType* to attach version numbers to each group to avoid unnecessary copying. We assume all version numbers attached to groups are positive. As usual with version numbers, we assume that they can be sufficiently large. We increment the version number of a group each time we modify at least one member in the group.

All schematic representations of segments of code that appear in Fig. 3 are the same as before, except

3. $com(\texttt{ref}\ X : nodeType;\ \texttt{in}\ g : 1..W,\ priv : bType;\ \texttt{out}\ tm : cType)$: performs an action on group g of the variable X of *nodeType* instead of on the whole object X.
4. $read(\texttt{ref}\ X : nodeType;\ \texttt{in}\ Y : nodeType,\ g : 1..W)$: only reads the value from group g of node Y to the same group of node X.

The relations corresponding to these schematic commands are adapted accordingly.

In the first implementation, *mp* becomes an array used to record pointers to private copies of shared nodes. In total we declare $N * P$ extra shared nodes for private use (one for each process and each node). Note that $node[mp[x].p]$ can be taken as a "private" node of process p though it is declared publicly. Array *indir* continues to act as pointers to shared nodes.

At the moment that process p reads group $i.p$ of $node[m.p]$ (line $l5$), process p may observe the object in an inconsistent state (i.e. the read value is not the current or historical view of the shared object) since pointer $m.p$ may have been redirected to some private copy of the node by some faster process q, which has increased the modified group's version number(in lines $l9$ and $l10$). When process p restarts the loop, it will get higher version numbers at the array *new*, and only needs to reread the modified groups, whose *new* version numbers differ from their *old* version numbers. Excessive copying can be therefore prevented. Line $l6$ is used to check if the read value of a group is consistent with the version number.

The first implementation is fast for an application that often changes only a small part of the object. However, the space complexity is substantial because $P+1$ copies of each node are maintained and copied back and forth. Sometimes, a trade-off is chosen between space and time complexity. We therefore adapt it to our second lock-free algorithm for large objects (shown in Fig. 3 also) by substituting all statements enclosed by $(* \ldots *)$ for the corresponding statements in the first version. As we did for small objects, we use only one extra copy of a node for each process in the second implementation.

In the second implementation, since the private copy of a node may belong to some other node, a process first initializes all elements of *old* to be zero (line $l1$) before accessing an object, to force the process to make a complete copy of the entire object for the first attempt. The process then only needs to copy part of the object from the second attempt on. The space complexity for our second

```
CONSTANT
  P = number of processes; N = number of nodes;
  W = number of groups;
  K = N + N * P;                     (*  II : K = N + P; *)
Type nodeType = record
    val: array [1..W] of valType;
    ver: array [1..W] of posnat;
  end
Shared Variables:
  pub: aType; node: array [1..K] of nodeType;
  indir: array [1..N] of 1..K;
Private Variables:
  priv: bType; pc: [l1..l11];
  x: 1..N; m: 1..K;
  mp: array [1..N] of 1..K;          (*  II : mp: 1..K;  *)
  new: array [1..W] of posnat; old: array [1..W] of nat;
  g: 1..W; tm, tm1: cType; i: nat;
Program:
      loop
  l1:   noncrit(pub, priv, tm, x);
        choose group g to be modified;
        old := node[mp[x]].ver;        (*  II : old := $\lambda$ (i:1..W): 0; *)
        (*  II : replace all ‘‘mp[x]’’ below by ‘‘mp’’ *)
        loop
  l2:     m := LL(indir[x]);
  l3:     i := 1
  l4:     while i $\leq$ W do
            new[i] := node[m].ver[i];
            if new[i] $\neq$ old[i] then
  l5:         read(node[mp[x]], node[m], i); old[i] := 0;
  l6:         if not VL(indir[x]) then goto l2; fi;
  l7:         node[mp[x]].ver[i] := new[i]; old[i] := new[i];
            fi;
            i++;
          end;
  l8:     if guard(node[mp[x]], priv) then
  l9:       com(node[mp[x]], g, priv, tm1); old[g] := 0;
            node[mp[x]].ver[g] := new[g]+1;
  l10:      if SC(indir[x], mp[x]) then
                mp[x] := m; tm := tm1; break;
            fi
  l11:    elseif VL(indir[x]) then break;
          fi
        end
      end
```

Fig. 3. Lock-free implementation I (* implementation II *) for large objects

version saves $(N-1) \times P$ times of size of a node, while the time complexity is more due to making one extra copy of the entire object for the first attempt. To see why these two algorithms are correct, we refer the interested reader to [7] for the complete mechanical proof.

6 Conclusions

This paper shows an approach to verification of simulation and refinement between a lower-level specification and a higher-level specification. It is motivated by our present project on lock-free garbage collection. Using the reduction theorem, the verification effort for a lock-free algorithm becomes simpler since fewer invariants are required and some invariants are easier to discover and easier to formulate without considering the internal structure of the final implementation. Apart from safety properties, we have also considered the important problem of proving liveness properties using the strong fairness assumption.

A more fundamental problem with Herlihy's methodology is the overhead that results from having multiple processes that simultaneously attempt to update a shared object. Since copying the entire object can be time-consuming, we present two enhanced algorithms that avoid unnecessary copying for large objects in cases where only small part of the objects are modified. It is often better to distribute the contents of a large object over several small objects to allow parallel execution of operations on a large object. However, this requires that the contents of those small objects must be independent of each other.

Formal verification is desirable because there could be subtle bugs as the complexity of algorithms increases. To ensure our hand-written proof presented in the paper is not flawed, we use the higher-order interactive theorem prover PVS for mechanical support. PVS has a convenient specification language and contains a proof checker which allows users to construct proofs interactively, to automatically execute trivial proofs, and to check these proofs mechanically. For the complete mechanical proof, we refer the reader to [7].

References

1. B. Bershad: Practical Considerations for Non-Blocking Concurrent Objects. In Proceedings of the 13th International Conference on Distributed Computing Systems, May 1993.
2. E.H. Jensen, G.W. Hagensen, and J.M. Broughton: A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livemore National Laboratory, November 1987.
3. E. Clarke, O. Grumberg, and D. Long: Model checking and abstraction ACM Transactions on Programming Languages and Systems 16(5), January 1994.
4. G. Barnes: A method for implementing lock-free data structures. In Proceedings of the 5th ACM symposium on Parallel Algorithms & Architecture, June 1993.
5. Henry Massalin, Calton Pu: A Lock-free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Columbia University, 1991

6. H. Gao, J.F. Groote and W.H. Hesselink.: Efficient almost wait-free parallel accessible dynamic hashtables. Technical Report CS-Report 03-03, Eindhoven University of Technology, The Netherlands, 2003. To appear in the proceedings of IPDPS 2004.
7. http://www.cs.rug.nl/˜wim/mechver/LLSCreduction
8. J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors: Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness. Lecture Notes in Computer Science 430. Spinger-Verlag, 1990.
9. Anthony LaMarca: A Performance Evaluation of Lock-free Synchronization Protocols. In proceedings of the thirteenth symposium on priniciples of distributed computing, 1994.
10. L. Lamport: The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems 16(3), 1994, pp. 872–923.
11. M. Abadi and L. Lamport: The existence of refinement mappings. Theoretical Computer Science, 2(82), 1991, pp. 253–284.
12. Mark Moir: Practical Implementations of Non-Blocking Synchronization primitives. In Proceedings of the sixteenth symposium on principles of Distributed computing, 1997. Santa Barbara, CA.
13. M. P. Herlihy: A methodology for implementing highly concurrent objects. ACM Transactions on Programming Languages and Systems 15, 1993, pp. 745–770.
14. Maurice Herlihy, Victor Luchangco and Mark Moir: The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In Proceedings of the 16th International Symposium on DIStributed Computing, 2002.
15. Victor Luchangco, Mark Moir, Nir Shavit: Nonblocking k-compare-single-swap. In Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms, 2003, pp. 314-323.
16. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag, 1992.