ELSEVIER

Information Processing Letters 68 (1998) 173–177

Information Processing Letters

Invariants for the construction of a handshake register

Wim H. Hesselink [1]

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

Received 23 July 1998; received in revised form 9 September 1998
Communicated by D. Gries

Abstract

Tromp's construction of a waitfree atomic register for one writing process and one reading process is presented and proved by means of ghost variables and invariants. Preservation of the invariants is proved mechanically. This approach can be compared with the original proof based on the partial order on the set of accesses of shared variables. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Waitfree; Handshake; Register; Invariant; Ghost variable; Distributed computing

1. Diachrony versus synchrony

In the field of distributed algorithms, correctness is often proved by means of operational arguments in terms of executions, i.e., sequences of accesses of shared variables. These accesses are assumed to take time. They are partially ordered by a relation $\rightarrow$ (precedes) such that $a \rightarrow b$ means that access a is completed before access b starts, cf. [5].

An alternative approach, cf. [6], concentrates on the properties of the states that may occur in the computations. Here ghost variables (also called history variables) are used to record relevant parts of the history in the state. The analysis then proceeds with invariants, and possibly variant functions.

The differences between the two approaches are described best by calling the first approach diachronous, and the second one synchronous. Coming from the synchronous school myself, I find it very hard to read (let alone understand) a diachronous argument. Yet it seems clear that the two schools can learn from each other.

To enable comparison I give here a synchronous treatment of an algorithm that comes with a diachronic proof, cf. [7].

2. Constructing a handshake register

A handshake register is a data structure shared by two concurrent processes, called Writer and Reader, such that Writer writes a value into the data structure and that Reader reads it. It must be guaranteed that any value read by Reader was at some time during the read action a recently written value (recentness) and that the results of subsequent read actions must occur in the same order as they were written (sequentiality). Finally, the handshake register must be waitfree, i.e., each of the processes Writer and Reader must be able to complete its action in a bounded number of steps, independently of the activities of the other process, cf. [2].

[1] Email: wim@cs.rug.nl. Web site: http://www.cs.rug.nl/~wim.

PII: S0020-0190(98)00158-6

Example. Assume that Writer writes the subsequent values $x0$ and $x1$, with $x0 \neq x1$, and that Reader reads the subsequent values $y0$ and $y1$. Moreover, let both read actions start after completion of the first write action and be completed during the second write action. Then recentness implies that $y0$ and $y1$ are elements of the set $\{x0, x1\}$. Sequentiality excludes the possibility $y0 = x1 \wedge y1 = x0$. The remaining three cases are allowed.

We describe a construction of a handshake register of a given type *Item*, based on four safe registers of type *Item* together with four Boolean atomic registers. Here, a register is said to be safe iff it guarantees that Reader reads the value most recently written, provided Writer has completed its write action before Reader started reading.

Tromp has proved that under certain assumptions on the shape of the algorithm, four safe registers are necessary for the construction of a handshake register. It is easy to argue that three are needed. Indeed, one register is used by Writer to write a new value, one register is needed to hold the value written most recently, and one register can be kept by the reading Reader. Since the handshake register must be waitfree, these roles cannot be combined.

Our construction is almost identical to Tromp's construction, cf. [7] (we found the algorithm more or less independently). So we use four safe registers, organized as a 2×2 array. The first Boolean index is modified whenever Reader catches up with Writer. The second Boolean index is modified whenever Writer completes a write action.

In general, waitfreedom does not imply efficiency, but this algorithm is also efficient. Its time complexity is almost the same as the time complexity of the constituent safe registers. As for applicability, the algorithm seems to be ideally suited, e.g., for a real-time system that has to react to a stream of measurement data.

We used the theorem prover NQTHM of Boyer and Moore, cf. [1], to verify that the invariants claimed are invariant. We also formed the global invariant of the system and proved that it is preserved under every step. The events file of this proof can be obtained as in [4]. In a complete mechanical proof of the algorithm, the next stage would be to prove that the global invariant is preserved in every execution sequence, but this is completely standard. Thus, in our set-up, the diachronous argument is rather trivial and comes on top of the synchronous analysis.

It seems to us that it would be harder to construct a proof for a theorem prover based on the diachronic approach. One should keep in mind, however, that a mechanical proof never solves the question how informal requirements are formalized in proof obligations. For instance, below, we introduce ghost variables to formalize assertions like recentness and sequentiality of the registers. If one uses the diachronous definitions, the diachronous stage of our mechanical proof would be longer. One could even choose to eliminate the ghost variables, i.e., prove that the assertions made remain valid for the algorithm obtained by removing the ghost variables.

Of course, it would be even easier to verify the correctness of the algorithm by means of a model checker. Such a verification, however, would not have uncovered the beautiful invariant (Lq0), obtained below.

3. Description of the algorithm

We use capitals for shared variables and lower case for private variables of the processes. Processes Writer and Reader are modeled as looping sequential processes with instruction pointers q and r, respectively. The variables used for data transfer are

x, y: *Item*;
Y: **array** *Bit* **of array** *Bit* **of** *Item*.

Here x is the value to be written, y is the value that is read, and Y is the array of the four safe registers. Array subscription is denoted by means of an infix dot, which binds to the left in order to allow carrying. For example, $Y.b.c$ would be written in Pascal as $Y[b][c]$ or $Y[b, c]$. We use $Bit = \{0, 1\}$ as synonymous with *Boolean*. The control variables used are the instruction pointers, q and r, and the Booleans

A, B, a, b, c: *Bit*;
C: **array** *Bit* **of** *Bit*.

The initial conditions are that all variables have values of the appropriate type and satisfy the initial predicate

$$q = 0 \wedge r = 0 \wedge B = b \wedge b = A.$$

The programs are

Writer
0. $get(x)$; $a := \neg B$;
1. $Y.a.(\neg C.a) := x$;
2. $C.a := \neg C.a$;
3. $A := a$; **goto** 0.

Reader
0. $b := A$;
1. $B := b$;
2. $c := C.b$;
3. $y := Y.b.c$; $put\ (y)$; **goto** 0.

The numbered instructions are treated as atomic commands with interleaving semantics. The fact that registers $Y.i.j$ are only assumed to be safe gives rise to an additional proof obligation, predicate (Jq0) discussed below.

It is well known that private computations or modifications of private variables may be combined atomically with actions on shared variables. We have done so in the instructions 1 and 0 of Writer and 3 of Reader. Moreover, since Reader does not modify array C, we allow Writer to read the shared variable $C.a$ together with access to other shared variables.

In Tromp's version, Writer has a private copy of array C, and there is an optimization that sometimes avoids execution of Writer's instruction 2. Moreover, Tromp avoids to assign to a shared Boolean variable its current value.

We use numbered instructions, instruction pointers, and **gotos** in order to be very precise about the atomicity of the instructions and about the program locations.

4. The proof of the algorithm

It is clear that, if it is correct, the algorithm is waitfree and very efficient. In fact, the entire task of writing or reading one value is performed in four instructions without waiting.

Since array Y consists of registers that are only supposed to be safe, we have the proof obligation that reading and writing at Y does not interfere. This is formalized in the predicate

(Jq0) $q = 1 \wedge r = 3 \wedge a = b \Rightarrow c = C.a$.

In fact, since q and r are the instruction pointers of Writer and Reader, respectively, this asserts that, when Reader is about to read register $Y.b.c$ and Writer is about to write register $Y.a.(\neg C.a)$, the two registers differ.

We prove that (Jq0) is an invariant of the system. If Reader executes the instruction at $r = 2$, it establishes $c = C.b$, and this implies (Jq0). If Writer executes the instruction at $q = 0$, it establishes $a = \neg B$, and then (Jq0) is implied by the obvious invariant

(Jq1) $r = 1 \vee B = b$.

The other two proof obligations are recentness and sequentiality. In order to formalize these concepts, the values written and read are numbered in order. These numbers serve as time stamps. For this purpose we introduce ghost variables declared by

t, U, u, s: $\mathbb{N}$;
S: **array** *Bit* **of array** *Bit* **of** $\mathbb{N}$,

with the initial condition

$s = u = U = S.i.j = t$.

Here t is the number of the value written most recently, U is the number of the latest write action completed, u is the number of the latest write action completed before the current read action, $S.i.j$ is the number of the value in $Y.i.j$, and s is the number of the value that has been read most recently.

To justify or formalize this informal description, we extend programs Writer and Reader with actions on the ghost variables in the following way (note that actions on ghost variables may be combined atomically with other actions).

Writer
0. $get\ (x)$; $a := \neg B$;
1. $Y.a.(\neg C.a) := x$;
   $t := t + 1$;
   $S.a.(\neg C.a) := t$;
2. $C.a := \neg C.a$;
3. $A := a$; $U := t$; **goto** 0.

Reader
0. $b := A$; $u := U$;
1. $B := b$;
2. $c := C.b$;
3. $y := Y.b.c$; $put\ (y)$;
   $s := S.b.c$; **goto** 0.

Now recentness and sequentiality are expressed in the subsequent proof obligations

(Kq0) $r = 3 \Rightarrow u \leqslant S.b.c$;
(Kq1) $r = 3 \Rightarrow s \leqslant S.b.c$.

Indeed, (Kq0) and (Kq1) express that the value y read at $r = 3$ is not older than the latest write action before the current read action, and also not older than the value that has been read most recently.

We shall prove that these predicates are invariants of the system. Note that B is still the only shared variable written by Reader.

In order to preserve predicates (Kq0) and (Kq1) when Reader executes the instruction at $r = 2$, we postulate the additional invariants

(Kq2) $u \leqslant S.b.(C.b)$;
(Kq3) $s \leqslant S.b.(C.b)$.

Preservation of (Kq0), (Kq1), (Kq2), and (Kq3) at $q = 1$ follows from the new postulate

(Kq4) $S.i.j \leqslant t$.

Preservation of (Kq2) and (Kq3) at $q = 2$ follows from (Kq4) and the new postulate

(Kq5) $q = 2 \Rightarrow t = S.a.(\neg C.a)$.

Preservation of (Kq2) and (Kq3) by the actions of Reader follows from the new postulates

(Kq6) $U \leqslant S.A.(C.A)$;
(Kq7) $S.b.(C.b) \leqslant S.A.(C.A)$;
(Kq8) $r = 3 \Rightarrow S.b.c \leqslant S.b.(C.b)$.

It was at this point in our originally handwritten proof that we felt the need for mechanical support. Array assignments are tricky if predicates contain arrays subscripted by elements of arrays.

It is easy to see that (Kq4) and (Kq5) are invariants. Preservation of (Kq6) at $q = 2$ and $q = 3$ follows from (Kq5) and

(Kq9) $U \leqslant t$;
(Kq10) $q = 3 \Rightarrow t = S.a.(C.a)$.

Preservation of (Kq7) at $q = 3$ follows from (Kq4) and (Kq10). Preservation of (Kq7) at $q = 2$ follows from (Kq4), (Kq5), and the new postulate

(Lq0) $(B = b \wedge b = A) \vee (A = a \wedge a = \neg B)$.

Preservation of (Lq0) is proved by means of the following beautiful argument. Consider the four equations $B = b$, $b = A$, $A = a$, $a = \neg B$. Among these four equations, the number of invalid ones is always odd. In the algorithm the equations are touched only by the assignments $B := b$, $b := A$, $A := a$, $a := \neg B$, each of which can invalidate at most one of these equations. Since the number of invalid ones is initially one, it remains one. Predicate (Lq0) is an invariant since it is equivalent to the assertion that precisely one of the equations is invalid. The theorem prover, of course, uses a straightforward case distinction to prove the invariance of (Lq0).

Predicate (Kq8) is preserved because of (Jq0) at $q = 1$, and (Kq4) and (Kq5) at $q = 2$. Preservation of (Kq9) is immediate. Preservation of (Kq10) follows from (Kq5), used at $q = 2$.

Note that we invent the invariants only as the need arises, for the specification or to prove preservation of other invariants. We prefer the invariants to be as weak as possible. For, if the invariants are too strong, they cannot be imposed initially.

The construction of the mechanical proof (see [4]) took only some eight hours. This was partially due to the fact that we could start from an NQTHM theory for distributed processes with shared memory and array assignments, cf. [3]. Also, we did not go beyond preservation of invariants.

5. Concluding remarks

We have given a synchronous proof of a variation of Tromp's construction of a waitfree atomic register for one writer and one reader. To us, our proof seems to be more perspicuous, but we leave the judgement to the unprejudiced reader.

Tromp's paper [7] proceeds by replacing the atomic Boolean variables by safe Boolean variables. To prove this variation by means of invariants is an easy exercise that yields no new insights.

References

[1] R.S. Boyer, J.S. Moore, A Computational Logic Handbook, Academic Press, Boston, 1988.
[2] M.P. Herlihy, Wait-free synchronization, ACM Trans. Program. Languages and Systems 13 (1991) 124–149.

[3] W.H. Hesselink, The design of a linearization of a concurrent data object, in: D. Gries, W.-P. de Roever (Eds.), Programming Concepts and Methods, Proceedings Procomet '98, Chapman & Hall, IFIP 1998, pp. 205–224.
[4] W.H. Hesselink, http://www.cs.rug.nl/~wim/crud/wfhandshake.events.
[5] L. Lamport, On interprocess communication, Parts I and II, Distrib. Comput. 1 (1986) 77–101.
[6] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Inform. 6 (1976) 319–340.
[7] J. Tromp, How to construct an atomic variable, in: J.-C. Bermond, M. Raynal (Eds.), Distributed Algorithms, Proceedings Nice, Lecture Notes in Comput. Sci., Vol. 392, Springer, Berlin, 1989, pp. 292–302.