Verifying Single and Multi-mutator Garbage Collectors with Owicki-Gries in Isabelle/HOL

Leonor Prensa Nieto* and Javier Esparza

Technische Universität München
Institut für Informatik, 80290 München, Germany
{prensani,esparza}@in.tum.de

Abstract. Using a formalization of the Owicki-Gries method in the theorem prover Isabelle/HOL, we obtain mechanized correctness proofs for two incremental garbage collection algorithms, the second one parametric in the number of mutators. The Owicki-Gries method allows to reason directly on the program code; it also splits the proof into many small goals, most of which are very simple, and can thus be proved automatically. Thanks to Isabelle's facilities in dealing with syntax, the formalization can be done in a natural way.

1 Introduction

The Owicki-Gries proof system [11] is probably the simplest and most elegant extension of Hoare-logic to parallel programs with shared-variable concurrency. Like Hoare-logic, it is a syntax oriented method, i.e., the proof is carried out on the program's text. Moreover, it provides a methodology for breaking down correctness proofs into simpler pieces: once the sequential components of the program have been annotated with suitable assertions, the proof reduces to showing that the annotation of each component is valid in Hoare sense, and that each assertion of an annotation is invariant under the execution of the actions of the other components (so-called interference-freeness). Finally, the annotated program helps humans to understand why the algorithm works, and to gain confidence in the proof.

One problem of the method is that the number of interference-freeness tests is $O(k^n)$, where n is the number of sequential components, and k is the maximal number of lines of a component. This makes a complete pencil and paper proof very tedious, even for small examples. For this reason, many of the interference-freeness proofs, which tend to be very simple, are usually omitted. This, however, increases the possibility of a mistake. One way out of this situation is to apply a theorem prover which automatically proves the easy cases, ensures that no mistakes are made, and guarantees that the proof is complete.

In [10], the Owicki-Gries method was formalized in the theorem prover Isabelle/HOL. In this paper we show that the method and its mechanization can be successfully applied to larger examples than those considered in [10]. We study

* Supported by the DFG PhD program "Logic in Computer Science".

two garbage collection algorithms. We first verify (a slightly modified version of) Ben-Ari's classical algorithm [2]. A pencil and paper proof using the Owicki-Gries method plus ad-hoc reasoning was presented in [14]. Our proof follows [14], but it manages to formulate the extra reasoning within the Owicki-Gries method. Ben-Ari's algorithm has also been mechanically proved using the Boyer-Moore prover [13] and PVS [6], but none of these proofs uses Owicki-Gries. This makes the algorithm an excellent example for comparing the Owicki-Gries method with others, and for comparing Isabelle/HOL with other theorem provers.

In the last section of the paper we verify a parametric garbage collector in which an arbitrary number of mutators work in parallel. The algorithm was proved by hand in [8] with the help of variant functions. To our knowledge this is the first mechanized proof. Notice that correctness must be shown for an infinite family of algorithms, which introduces an additional difficulty.

The paper is structured as follows: in Section §2 we briefly present our language, the Owicki-Gries method and some basic information about Isabelle/HOL. The basics of garbage collection algorithms are described in Section §3. Section §4 presents the proof of Ben-Ari's algorithm in detail. Section §5 presents the proof of the parametric algorithm. Section §6 contains conclusions.

For space reasons we only sketch the proof of the parametric algorithm, and sligthly simplify the annotations of the programs. Complete annotations and proof scripts can be obtained from `http://www.in.tum.de/~prensani/`.

2 The Owicki-Gries Method and Isabelle/HOL

The Owicki-Gries proof system is an extension of Hoare-logic to parallel programs. Two new statements deal with parallel processing. The `COBEGIN-COEND` statement encloses processes that are executed in parallel; the `AWAIT` statement provides synchronization. We consider the evaluation of any expression or execution of any assignment as an *atomic action*, i.e., an indivisible operation that cannot be interrupted. If several instructions are to be executed atomically, they form an *atomic region*. Syntactically, these are enclosed in angled brackets $<$ and $>$.

Proofs for parallel programs are given in the form of *proof outlines*, i.e., the program is annotated at every control point where interference may occur. Given two proof outlines S and T, we say that they are *interference free* if, for every atomic action s in S with precondition $pre(s)$, and every assertion P in T, the formula $\{P \wedge pre(s)\}\ s\ \{P\}$ holds, and conversely. Thus, the execution of any atomic action cannot affect the truth of the assertions in the parallel programs. The inference rule for the verification of the `COBEGIN-COEND` statement is:

$$\frac{\{p_i\}S_i\{q_i\}\ \textit{for}\ i \in \{1,\ldots,n\}\ \textit{are correct and interference free}}{\{\bigwedge_{i=1}^{n} p_i\}\ \texttt{COBEGIN}\ S_1 \| \ldots \|\ S_n\ \texttt{COEND}\ \{\bigwedge_{i=1}^{n} q_i\}}$$

An important aspect of the Owicki-Gries method is the use of auxiliary variables. They augment the program with additional information for proof purposes. An auxiliary variable a is only allowed to appear in assignments of the form $a := t$, and so it is superfluous for the real computation.

Isabelle [1, 12] is a generic interactive theorem prover and Isabelle/HOL is its instantiation for higher-order logic. For a tutorial introduction see [9]. We do not assume that the reader is already familiar with HOL and summarize the relevant notation: The ith component of the list xs is written $xs\,!\,i$, and $xs\,[i := x]$ denotes xs with the ith component replaced by x. Set comprehension syntax is $\{e.\ P\}$. To distinguish variables from constants, we show the latter in sans-serif. We will use the syntax for while-programs as it is formalized in Isabelle/HOL: Assertions are surrounded by "{." and ".}". The syntax for assignments is $x ::= t$. Sequential composition is represented by a double semi-colon (;;) or by a double comma (,,) when it occurs inside atomic regions.

3 Garbage Collection

Garbage collection is the automatic reclamation of memory space. User processes, called *mutators*, might produce garbage while performing their computations. The *collector*'s task is to identify this garbage and to recycle it for future use by appending it to the *free list*. *Incremental* (also called on-the-fly) garbage collection systems, are those where the garbage collection work is randomly interleaved with the execution of instructions in the running programs.

The **memory** is modelled as a finite directed graph with a fixed number of nodes, where each node has a fixed set of outgoing edges. A pre-determined subset of nodes, called the Roots, is always accessible to the running program. A node is called *reachable* or *accessible* if a directed path exists along the edges from at least one root to that node, otherwise, it is called *garbage*. For marking purposes, each node is associated a color, which can be black or white. The memory structure can only be modified by one of the following three operations: redirect an edge from a reachable node towards a reachable node, append a garbage node to the free list, or change the color of a node.

The **mutators** abstractly represent the changes that user programs produce on the memory structure. It is assumed that they only work on nodes that are reachable, having the ability to redirect an edge to some new target. To make garbage collection safe, the mutators cooperate with the collector by assuming the overhead of blackening the new target. Thus, a mutator repeatedly redirects some edge `R` to some reachable node `T`, and then colors the node `T` black.

It is customary to describe the **collector**'s task in this way: identify the nodes that are garbage, i.e., no longer reachable, and append them to the free list, so that their space can be reused by the running program. However, at an abstract level it suffices to assume that the collector makes garbage nodes accessible again: since the mutator has the ability to redirect arbitrary accessible edges, it may reuse these nodes. In the sequel adding a node to the free list will just mean making it accessible.

The collector repeatedly executes two phases, traditionally called "marking phase" and "sweep" or "appending phase". In the marking phase the collector (1) colors the roots black; (2) visits each edge, and if the source is black it colors the target black; (3) counts the black nodes; (4) if not all reachable nodes are black, goes to step (2). In the appending phase, the collector (5) visits each node, appending white nodes to the free list, and coloring black nodes white. The safety property we prove says that *no reachable node is garbage collected*. In other words, if during the appending operation a node is white, then it is garbage. Clearly, this property holds if step 4 is correct. But how do we determine that all reachable nodes are black? In the case of one mutator, Ben Ari's solution is to keep the result of the last count, and compare it with the result of the current count. If they coincide, then all reachable nodes are black. For n mutators, we compare the results of the last $n+1$ counts. So the algorithms for one and several mutators differ only in step 4.

4 The Single Mutator Case

We verify (a slightly modified version of) Ben-Ari's algorithm. We follow the ideas of [14], but formulate the proof completely within the Owicki-Gries system.

The Memory. The memory is formalized using two lists of fixed size. In the first list, called M, memory nodes are indexed by natural numbers that range from 0 to the length of M; the color of node i can be consulted by accessing M!i. The second list, called E, models the edges; each edge is a pair of natural numbers corresponding to the source and the target nodes. Roots is an arbitrary set of nodes. Reach is the set of nodes reachable from Roots (including Roots itself). Blacks is the set of nodes that are Black . Finally, BtoW are the edges that point from a Black node to a White node.

The separate treatment of colors and edges in our data structure is an abstraction that considerably simplifies proofs relating to the changes in the graph. If an edge is redirected, M remains invariant, while coloring does not modify E.

The Mutator. The auxiliary variable z is false if the mutator has already redirected an edge but has not yet colored the new target. Some obvious conditions required of the selected edge R and node T, namely, R<length E and T<length M always hold and are omitted in the annotated program text. The verification requires to prove one lemma: an accessible node cannot be rendered inaccessible by redirecting an edge to it.

```
{.T ∈ Reach E ∧ z.}
WHILE True INV {.T ∈ Reach E ∧ z.}
DO < E::=E[R:=(fst(E!R),T)],, z::=¬ z >;;
   {.T ∈ Reach E ∧ ¬ z.}
   < M::=M[T:=Black],, z::=¬ z >
OD {.False.}
```

Fig. 1. The mutator

The Collector. The collector first blackens the roots and then executes a loop. The body of the loop consists of first traversing M coloring all reachable nodes black, and then counting the number of black nodes. The loop terminates if the results of the current count and the previous one coincide. After termination of the loop, the collector traverses M once more, this time making all white nodes reachable and all black nodes white. We divide the algorithm into modules, which are pieces of code together with their pre- and postconditions. The Blackening_Roots module is straighforward; the codes and annotations of the rest are explained separately. Obvious intermediate assertions are omitted.

```
{.True.}
WHILE True INV {.True.}
DO Blackening_Roots;;
   {.Roots ⊆ Blacks M.}
   OBC::={};; BC::=Roots;; Ma::=L;;
   WHILE OBC ≠ BC
    INV {. Roots ⊆ Blacks M
         ∧ OBC ⊆ Blacks Ma ⊆ BC ⊆ Blacks M
         ∧ (Safe(M,E) ∨ OBC ⊂ Blacks Ma ).}
   DO OBC::=BC;; Propagating_Black;;
      Ma::=M;; BC::={};; Counting
   OD;;
  {.Safe(M,E).}
  Appending
OD {.False.}
```

Fig. 2. The collector

Safe(M,E) states that all reachable nodes are black, i.e., Reach E ⊆ Blacks M. Since we have Safe(M,E) before Appending, all white nodes are garbage right before the appending module starts. This is almost the safety property we wish to prove, since, as we shall show later when describing the Appending module, if a white node is garbage before Appending, then it remains so until Appending makes it reachable.

The variables BC (Black Count) and OBC (Old Black Count) are used to determine if the set of black nodes has grown during the last Propagating_Black phase. Following [14], OBC is initialized to $\{\}$, and BC to the set Roots[1]. A single auxiliary variable Ma is used for "recording" the value of M after the execution of Propagating_Black. The constant L is used to give Ma a suitable first value, defined as a list of nodes where only the Roots are black.

```
{.Roots ⊆ Blacks M
 ∧ OBC ⊆ BC ⊆ Blacks M.}
I::=0;;
WHILE I<length E
 INV {. Roots ⊆ Blacks M
       ∧ OBC ⊆ BC ⊆ Blacks M
       ∧ PB(M,E,OBC,I,z)
       ∧ I ≤ length E.}
DO IF M!(fst(E!I))=Black THEN
      M::=M[snd(E!I):=Black] FI;;
   I::=I+1
OD
{.Roots ⊆ Blacks M
  ∧ OBC ⊆ BC ⊆ Blacks M
  ∧ (OBC ⊂ Blacks M ∨ Safe(M,E)).}
```

Fig. 3. Module Propagating_Black

The key parts of the invariant are the second and third conjuncts. The second conjunct guarantees that after any execution of the body the cardinalities of OBC and BC are a lower and upper bound, respectively, of the number of black nodes after Propagating_Black. (It is clear that OBC is a lower bound, because black nodes stay black until the beginning of the appending phase. That BC is an upper bound is the difficult part, since the mutator can blacken nodes while the collector executes Counting.) The third conjunct guarantees that, if an execution of the body does not establish the safety property, then OBC is a proper lower bound, which means that some white node was colored black during the execution of Propagating_Black. The Propagating_Black and Counting modules have very clear tasks: Propagating_Black establishes the third conjunct, while Counting establishes the second.

[1] OBC and BC are here sets of black nodes whereas in the original algorithm they represent their cardinalities. We found the set approach easier to formalize but it simplifies neither the algorithm nor the proofs.

Propagation of the Coloring. During this phase, the collector visits the edges in a given order, coloring the target whenever the source was Black. This phase establishes the third conjunct of the invariant.

The invariant of this module is tricky. The predicate PB is an adaptation of the one proposed in [14]. PB(M,E,OBC,I,z) denotes the predicate

```
OBC ⊂ Blacks M ∨ (∀ i<I. ¬ BtoW(E!i,M)
∨ (  ¬ z ∧ i=R ∧ snd(E!R)=T ∧ (∃ r<length E. I ≤ r ∧ BtoW(E!r,M) )))
```

and it is the crux of the proof. Intuitively, its invariance is proved as follows. If the collector or the mutator blacken some white node, then after execution of the body OBC ⊂ Blacks M holds. If all the edges visited by the collector point to a Black node, then ∀ i<I. ¬ BtoW(E!i,M) holds. If some visited edge points to a white node (because the mutator has redirected it), then (and this is Ben-Ari's main observation) there is another BtoW edge among those that have not yet been visited: since the white node T is reachable, there is a path to T from some root, and since all roots are Black, some edge along this path must be a BtoW edge. Observe that upon termination of the loop this last clause cannot hold since I=length E. To obtain the postcondition OBC⊂Blacks M ∨ Safe(M,E) we need to prove a lemma: if all Roots are Black and no edge points from a Black node to a White node, then all reachable nodes are Black.

```
{.Roots ⊆ Blacks M
  ∧ OBC ⊆ Blacks Ma ⊆ Blacks M
  ∧ BC={}
  ∧ (OBC ⊂ Blacks Ma ∨ Safe(M,E)).}
I::=0;;
WHILE I<length M
 INV {.Roots ⊆ Blacks M
      ∧ OBC ⊆ Blacks Ma ⊆ Blacks M
      ∧ BC ⊆ Blacks M
      ∧ (OBC ⊂ Blacks Ma ∨ Safe(M,E))
      ∧ {i. i<I ∧ Ma!i=Black} ⊆ BC
      ∧ I ≤ length M.}
DO IF M!I=Black THEN BC::=(insert I BC) FI;;
   I::=I+1
OD
{.Roots ⊆ Blacks M
  ∧ OBC ⊆ Blacks Ma ⊆ BC ⊆ Blacks M
  ∧ (OBC ⊂ Blacks Ma ∨ Safe(M,E)).}
```

Fig. 4. Module Counting

Counting Black Nodes. This phase finally re-establishes the invariant of the collector's outermost loop. The computed set BC must contain all nodes which were black *upon termination of* Propagating_Black, or, since Ma records precisely this set, the Counting phase must ensure that Blacks Ma ⊆ BC holds. Since the mutator can only blacken nodes, this task is now trivial. With all reachable nodes marked we can proceed to the appending phase where all unmarked nodes are appended to the free list.

```
{.Safe(M,E).}
I::=0;;
WHILE I<length M
 INV {.Safe_I(M,E,I) ∧ I ≤ length M.}
DO IF M!I=Black THEN M::=M[I:=White]
   ELSE {.I∉Reach E ∧ Safe_I(M,E,I).}
         E::=AppendtoFree(I,E)
   FI;; I::=I+1
OD {.True.}
```

Fig. 5. Module AppendtoFree

Appending to the Free List. Here we follow our predecessors: Appending a garbage node I to the free list (i.e. making I reachable) is modelled by an abstract function AppendtoFree satisfying suitable axioms. In the annotated code, Safe_I(M,E,I) states that all white nodes with index I or larger are garbage. The precondition of the assignment to E guarantees that only garbage nodes are collected (the conjunct Safe_I(M,E,I) is needed here to maintain the invariant throughout the loop).

5 The Multi-mutator Case

If we allow the interaction with several mutators, new difficulties come into play. We consider a solution, first presented in [8], in which the collector proceeds to the appending phase only after n+1 consecutive executions of the Propagating_Black phase during which the set of black nodes did not increase. Observe that in the case of one mutator this collector checks *twice* whether OBC=BC, and not only once, as the collector of Section §4. In [8] it is shown that n consecutive executions suffice, but we do not consider this version in the paper.

The program consists of a fixed, finite and nonempty set of mutator processes and one collector process. When the number of programs is a parameter, the list of programs to be executed in parallel can be expressed using the function map and the construct [i..j], which represents the list of natural numbers from i to j (the syntax [i..j(] corresponds to [i..j-1]).

The syntax and the tactic for the generation of the verification conditions presented in [10] have been extended to deal with this kind of program schemas. They are preceeded by the word SCHEME.

```
SCHEME map [λ j.
{.Z (Muts!j).}
WHILE True
 INV {.Z (Muts!j).}
DO < IF T (Muts!j) ∈ Reach E THEN
        E::=E[R (Muts!j):=(fst(E!(R (Muts!j))),
              T (Muts!j))] FI ,,
     Muts::=Muts[j:=(Muts!j) (|Z:=False|)] > ;;
   {.¬ Z (Muts!j).}
   < M::=M[T (Muts!j):=Black] ,,
     Muts::=Muts[j:=(Muts!j) (|Z:=True|)] >
OD {.False.} )
[0..n(]
```

Fig. 6. The mutators

The Mutators. A mutator can only redirect an edge when its target is a reachable node, and redirecting may make its old target inaccessible. If several mutators are active, then one of them may select a reachable node T as new target, but another one may render T inaccessible before the edge has been redirected to T. To solve this problem, selecting the new target and redirecting the edge is modelled as a single atomic action.

Each mutator m selects an edge R_m and a target node T_m. As in the previous section each mutator owns an auxiliary variable Z_m that indicates when the mutator is pending before the blackening of a node. These three objects are put together in a record. Isabelle's syntax for accessing the field Z of a record variable Mut is Z Mut. Record update is written Mut (|Z:=True|), meaning that the field Z of the record Mut is updated to the value True. The variable Muts is a list of length n (the number of mutators) whose components are records of type mut. For example, to access the selected edge of mutator j we write R (Muts!j).

The Collector. In the case of one mutator, if an execution of the body does not establish the safety property, then some white node was colored black during the execution of Propagating_Black. When several mutators are present, there may be other reasons. To describe them we need a new value Queue(Muts,M) which represents the number of mutators that are *queueing* to blacken a white node.

```
{.True.}
WHILE True INV {.True.}
DO Blackening_Roots;;
   OBC::={};; BC::=Roots;; l::=0;;
   WHILE l<n+1
    INV {. Roots ⊆ Blacks M
         ∧ OBC ⊆ BC ⊆ Blacks M
         ∧ (Safe(M,E)
             ∨ ( (BC ⊂ Blacks M
                   ∨ l ≤ Queue(Muts,M))
                ∧ l<n+1)).}
   DO  OBC::=BC;; Propagating_Black;;
       < Ma::=M,,Qa::=Queue(Muts,M) >;;
       BC::={};; Counting;;
       {. Roots ⊆ Blacks M
         ∧ OBC ⊆ Blacks Ma ⊆ BC ⊆ Blacks M
         ∧ (Safe(M,E)
             ∨ OBC ⊂ Blacks Ma
             ∨ (l<Qa ∧ OBC ⊂ Blacks M)
             ∨ (l<Qa ∧ Qa ≤ Queue(Muts,M)))
         ∧ Qa<n+1.}
       IF OBC=BC THEN l::=l+1 ELSE l::=0 FI
   OD;;
  {.Safe(M,E).}
  Appending
OD {.False.}
```

Fig. 7. The collector

The auxiliary variable Qa will "record" this value upon termination of the Propagating_Black phase. The invariant of the one mutator case must be compared with the precondition of the IF-THEN-ELSE instruction, because both correspond to the assertion established by the Counting phase. The assertion Safe(M,E) ∨ OBC ⊂ Blacks Ma has been weakened with new disjuncts, corresponding to the new situations which can prevent Safe(M,E) from holding. The first new disjunct corresponds to the case in which at least one mutator joins Queue(Muts,M) and then colors its new target during Counting. The second conjunct corresponds to the case in which no mutator performs any coloring. Intuitively, after n+1 non-blackening Propagating_Black iterations, the property Safe(M,E) must hold, since the number of queueing mutators cannot exceed n.

The codes of the modules are the same as in §4 up to annotations in the Propagating_Black and Counting phases, which have to be adapted to the new invariant.

We just show the invariant of the Propagating_Black phase:

```
{. Roots ⊆ Blacks M ∧ OBC ⊆ BC ⊆ Blacks M ∧ I ≤ length E.
  ∧ (   Safe(M,E) ∨ OBC ⊂ Blacks M ∨ l<Queue(Muts,M)
      ∨ (∀ i<I. ¬ BtoW(E!i,M) ∧ l ≤ Queue(Muts,M))) }
```

Any coloring establishes OBC ⊂ Blacks M. (Observe that only coloring can make the queue shorter.) If no coloring occurs then either all the visited edges point to a black node, or some mutator has redirected an edge to a white source but has not yet colored the target, which amounts to saying that the queue grows (l<Queue(Muts,M)).

6 Conclusions and Related Work

The Owicki-Gries method splits the proof into a large number of simple interference-freeness subproofs. These are very tedious to prove by hand, and so avoided by humans, who prefer to split a proof into a few difficult cases. In order to investigate if the use of a theorem prover can palliate this problem, we have provided mechanically checked Owicki-Gries proofs for two garbage collection algorithms. The result is: 320 out of 340 interference-freeness proofs in the final annotations were automatically carried out by Isabelle/HOL. For the remaining 20 interference-freeness proofs only three lemmas had to be supplied. The proofs of these lemmas, however, were very interactive.

We do not know of any complete Owicki-Gries proof for any of the two algorithms. In his proof of Ben-Ari's algorithm [14], van de Snepscheut mixes the Owicki-Gries method with ad-hoc reasoning; in particular, he does not provide an invariant for the outermost loop, implicitly claiming that doing so will be complicated. However, the invariant turns out to be simple (3 clauses), and has a clear intuitive interpretation. In [8], Jonker argues that " A proof [of the n-mutators algorithm] according to the Owicki-Gries theory would require the introduction of a satisfactory number of ghost variables .... In an earlier version of this paper the invariant we constructed was rather unwieldy and the proof of invariance almost unreadable." However, our proof only uses two auxiliary variables (`Ma` and `Qa`), plus a trivial auxiliary variable for each mutator. Extending our proof to the more elaborated n-mutator algorithms of [8] should be possible with reasonable effort.

We know of two other mechanized proofs of Ben-Ari's algorithm, carried out using the Boyer-Moore theorem prover [13] and PVS [6, 7]. The main advantage of our approach is probably the closeness to the original program text, which simplifies the interaction with the prover: Annotated programs are rather readable by humans, and they are also directly accepted as input by Isabelle. In other approaches the program must be first translated into a different language (e.g. LISP in [13]).

Another aspect of our formalization is that we only had to prove 8 lemmas (3 of them trivial) about graph functions, whereas 100 lemmas were required in [13], and about 55 in [6, 7]. The reason for this is that many trivial lemmas about sets or lists could be automatically proved using Isabelle's built-in tactics (rewriting, classical reasoning, decision procedures for Presburger arithmetic, etc) and Isabelle's standard libraries. The proof effort, however, took two months for the one-mutator algorithm (similar to our predecessors) and another two months for the n-mutator case. Most of the time was consumed in finding and improving the invariants.

A disadvantage of the Owicki-Gries method (in its classical version) is that it can only be applied to safety properties, while in [8, 13, 14] the liveness property "every garbage node is eventually collected" is also proved to hold.

None of our two algorithms has been proved correct using fully automatic methods. In [3] there is a proof of Ben Ari's algorithm for 1 mutator and 4 memory cells. In [4], a predecessor of Ben Ari's algorithm is proved correct using automatic tools for generating and proving invariants. The key invariants, however, require intelligent input from the user. The paper suggests using predicate abstraction for checking or strengthening invariants in a larger verification effort involving interactive theorem provers, which is a promising idea.

Our overall conclusion is that the application of a theorem prover greatly enhances the applicability of the Owicki-Gries method. The closeness to the original program is preserved, and the large number of routine proofs is considerably automatized.

References

1. Isabelle home page. `www.cl.cam.ac.uk/Research/HVG/isabelle.html`.
2. M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Toplas*, 6:333–344, 1984.
3. G. Bruns. *Distributed Systems Analysis with CCS* Prentice-Hall, 1997.
4. S. Das, D. L. Dill and S. Park. Experience with predicate abstraction. In CAV '99, LNCS 1633, 160-171, 1999.
5. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.
6. K. Havelund. Mechanical verification of a garbage collector. FMPPTA'99. Available at `http://ic-www.arc.nasa.gov/ic/projects/amphion/people/havelund/`.
7. K. Havelund and N. Shankar. A mechanized refinement proof for a garbage collector. *Formal Aspects of Computing*, 3:1-28, 1997.
8. J. E. Jonker. On-the-fly garbage collection for several mutators. *Distributed Computing*, 5:187–199, 1992.
9. T. Nipkow. *Isabelle/HOL. The Tutorial*, 1998. Unpublished Manuscript. Available at `www.in.tum.de/~nipkow/pubs/HOL.html`.
10. T. Nipkow and L. Prensa Nieto. Owicki/Gries in Isabelle/HOL. In FASE'99, LNCS 1577, 188–203. Springer-Verlag, 1999.
11. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
12. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, LNCS 828 Springer-Verlag, 1994.
13. D. M. Russinoff. A mechanically verified garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.
14. J. L. A. van de Snepscheut. "Algorithms for on-the-fly garbage collection" revisited. *Information Processing Letters*, 24:211–216, 1987.