What is a Garbage Collector? An Exercise in Compositional Refinement*

Klaus Havelund[1]** and Natarajan Shankar[2]***

[1] Jet Propulsion Laboratory, California Inst. of Technology, Pasadena, USA
[2] SRI Computer Science Laboratory, Menlo Park, USA

Abstract. Specifications define the expected behavior of software components and systems. They are critical to software correctness yet writing good specifications can be quite challenging. We examine the case of the specification for a concurrent garbage collector that operates in conjunction with a cooperative mutator. We argue that many previous attempts to specify the behavior of such a garbage collector are flawed. The typical problem is that correctness is specified in terms of assertions that hold at specific program points leaving a step of interpretation to be convinced that the intended behavior has been properly formalized. We introduce a notion of compositional refinement that serves as an acceptable specification for properties of components like a garbage collector that are refined in the context of an assumed environment like a mutator.

1 Introduction

Cliff Jones is celebrated for his work on Rely/Guarantee compositional verification, data refinement, and software specification particularly in the context of the Vienna Definition Method (VDM). We present a treatment of garbage collection that integrates all of these strands. Though there have been many attempts at specifying and proving garbage collectors, we argue that they all have subtle flaws. We introduce a notion of *compositional refinement* that remedies these flaws in both the specification and the verification of garbage collectors, particularly those that require cooperation from mutators. The compositional refinement technique is applicable in other contexts where component properties are established under an assumed environment and then composed to derive system properties.

A specification constrains the observable behavior of a software component. There are two purposes for software property specification. One is for *composition*, to ensure that components can be composed and that the specification for the composition emerges from the combination of the component specifications. For example, the Hoare triple [21] $\{P\}S\{Q\}$ constrains the behavior of

* © 2024. All rights reserved.

** The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

*** The research is supported by NSF:CCRI Award #2016597.

component S so that when it is executed on a state satisfying P it either diverges or terminates in a state satisfying Q. The composition rule then allows $\{P\}S_1; S_2\{Q\}$ to be derived from $\{P_1\}S_1\{Q_1\}$ and $\{P_2\}S_2\{Q_2\}$ if $P \Rightarrow P_1$, $Q_1 \Rightarrow P_2$, and $Q_2 \Rightarrow Q$. The other purpose is for *refinement* so that a component can be realized by specifications that are closer to being implementable in code. Thus, $\{P\}S\{Q\}$ is refined by $\{\underline{P}\}\underline{S}\{\underline{Q}\}$ if both triples are valid and $P \Rightarrow \underline{P}$ (precondition weakening) and $\underline{Q} \Rightarrow Q$ (postcondition strengthening). This implies that the statement S can be replaced by $\underline{S}$ in the context of the triple $\{P\}S\{Q\}$.[3] The refinement approach to correct-by-construction program derivation is developed in the refinement calculus [3,7,6,4,29,2,13].

With concurrent programs interacting through shared memory, the composition operator is $S_1 \| S_2$. Owicki and Gries [32] presented a proof outline logic for composing the proofs (annotated programs) for S_1 and S_2 such that each atomic statement in S_1 (under its precondition) preserves the assertions on atomic statements used in the proof for S_2, and vice-versa. Then, S_1 and S_2 are *non-interfering*, and we can conclude $\{P_1 \wedge P_2\}S_1 \| S_2\{Q_1 \wedge Q_2\}$ from $\{P_1\}S_1\{Q_1\}$ and $\{P_2\}S_2\{Q_2\}$.

Jones [23] presented a rule for a language that has sequential and concurrent composition that adds a reflexive Rely relation R and a reflexive guarantee relation G such that the Jones quintuple $[R]\{P\}S\{Q\}[G]$ captures the claim that program S when started in an initial state satisfying P and interleaved with environment transitions satisfying R, ensures post-condition Q and that each S transition satisfies G. The concurrent composition $S_1 \| S_2$ can then be verified using the rule:

$$[R_1 \wedge R_2]\{P_1 \wedge P_2\}S_1 \| S_2\{Q_1 \wedge Q_2\}[G_1 \vee G_2]$$

follows from

1. $[R_1]\{P_1\}S_1\{Q_1\}[G_1]$
2. $[R_2]\{P_2\}S_2\{Q_2\}[G_2]$
3. $G_1 \Rightarrow R_2$
4. $G_2 \Rightarrow R_1$

A different view of concurrent programs emerged in the form of transition systems with temporal specifications in the work of Pnueli [35], Lamport [26], Back and Kurki-Suonio [5], and Chandy and Misra [11]. These approaches dispense with precondition/postcondition specifications in favor of those that constrain infinite traces of behavior. Temporal logic [35] extends propositional logic with modal operators such as $\Box P$ (P is henceforth true), $\Diamond P$ (P is eventually true),

[3] The partial correctness refinement ordering allows the concrete statement $\underline{S}$ to diverge where the abstract statement S converges. The total correctness variant would require $wp(S, Q) \Rightarrow wp(\underline{S}, Q)$ so that $\underline{S}$ converges whenever S does. If there is a data refinement step in deriving the concrete representation, then if f maps concrete states to their abstract counterparts, the corresponding proof obligation can be stated as $wp(S, Q) \circ f \Rightarrow wp(\underline{S}, Q \circ f)$.

$P \mathcal{U} Q$ (Q is eventually true, but P holds until then), $P \mathcal{R} Q$ (P becomes true before Q becomes false, or P becoming true releases Q to no longer hold).

The UNITY framework [11] combines a guarded command language (with fair execution of the commands) and a lightweight temporal logic with temporal modalities such as **stable** p indicating that once p holds it continues to hold, and $p \leadsto q$ indicating that whenever p holds, q eventually holds along any fair execution. Refinement is performed by superposition, i.e., by defining a representation invariant that defines the abstract variables in terms of the concrete ones as a way of demonstrating any observable abstract behavior has a concrete counterpart.

Lamport's Temporal Logic of Actions (TLA) [26] uses state functions f and two-state assertions (actions) A to define programs and properties. An action predicate of the form $[A]_f$, defined as $A \vee f = f'$ and $\langle A \rangle_f$, defined as $A \wedge f' \neq f$. Programs are defined as temporal formulas of the form $I \wedge \Box[A]_f \wedge F$, where I is the initial state predicate, $[A]_f$ is a two-state action predicate asserting that in each step, either A holds or the state function f remains unchanged, and F is a fairness condition. In TLA, existential quantification over state variables is used for hiding internal state, and state mappings from the concrete state to the abstract state serve as refinement maps.

As noted, the refinement approach can be used to derive concurrent programs from abstract transition system specifications. A transition system is a triple $\langle \Sigma, I, N \rangle$ consisting of a state type Σ, an initialization predicate I, and a transition relation N. Additionally, we allow an observation state function π that maps Σ to an observable state type $\widehat{\Sigma}$. A behavior of the system is a sequence $\overline{\sigma}$ of states in Σ such that $I(\overline{\sigma}_0)$ holds and $N(\overline{\sigma}_i, \overline{\sigma}_{i+1})$ holds for all $i \geq 0$. An invariant P of a transition system is a predicate on Σ that holds of any σ that appears in any behavior of the transition system, i.e., $I \wedge \Box[A]_f \Rightarrow \Box P$.

Given two transition systems T_1 of the form $\langle \Sigma_1, \widehat{\Sigma}, I_1, N_1, \pi_1 \rangle$ and T_2 of the form $\langle \Sigma_2, \widehat{\Sigma}, I_2, N_2, \pi_2 \rangle$, and invariants P_1 of T_1 and P_2 of T_2, a refinement map ρ maps Σ_2 to Σ_1 such that

1. Observational identity: $\pi(\rho(\sigma)) = \pi(\sigma)$ for $\sigma \in \Sigma_2$, i.e., the observable state in the concrete and abstract systems must be identical.
2. Initiality: $I_2(\sigma) \Rightarrow I_1(\rho(\sigma))$ for $\sigma \in \Sigma_2$, i.e., The concrete initial states must be a subset of the abstract ones under the refinement map.
3. Simulation: $P_1(\rho(\sigma)) \wedge P_2(\sigma) \wedge P_2(\sigma') \wedge N_2(\sigma, \sigma') \Rightarrow N_1(\rho(\sigma), \rho(\sigma'))$, for $\sigma, \sigma' \in \Sigma_2$, i.e., the abstract transition under the refinement map must be able to simulate a concrete transition where the concrete invariant P_2 and abstract invariant P_1 (under ρ) hold on the prestate σ, and P_2 holds of the post-state σ'.

These conditions ensure that any valid observable behavior of T_2 is also a valid observable behavior of T_1. It might be necessary to augment T_2 with history and prophesy variables in order to construct a refinement map. Transition system specifications can also include fairness constraints F. For our purpose, we will assume that each fairness constraint F must hold infinitely often in any fair behavior. The refinement proof must demonstrate a well-founded ordering on the non-F_1 abstract states given the invariants and fairness constraints on the concrete transition system T_2.

A concurrent garbage collector poses an interesting challenge for both formalization and verification. A garbage collector is a program that operates in conjunction with a client program, a mutator. The *mutator* can access certain nodes on the heap, namely those reachable from the roots (the program variables) by dereferencing pointers, it can allocate nodes from the free list, and its actions can make some of these reachable nodes unreachable (garbage). The *collector* collects garbage heap nodes into the free list. There is a long history of progress and setbacks in formal treatments of garbage collection. Since proofs of algorithms like garbage collectors can be quite hard, it helps if one is proving the right theorem.

Garbage collection was first introduced by McCarthy with Lisp [28]. Most early garbage collectors were nonconcurrent, i.e., executed atomically when the free list was empty or nearly empty. We focus here on concurrent garbage collectors where the collector and mutator operate concurrently. The mutator allocates from the free list and creates garbage in the form of unreachable nodes through mutation steps that redirect pointer edges. The collector moves these unreachable nodes into the free list to make them reachable. Informally, the correctness criterion for such concurrent garbage collector is that all unreachable nodes are eventually made reachable through the free list without affecting the shape of reachable nodes that are not in the free list.

Dijkstra, Lamport, Martin, Scholten, and Steffens [14] (henceforth abbreviated as DLMSS) gave the first formal treatment of a concurrent garbage collector. Gries [18] gave an invariance proof using the Owicki–Gries method [32]. The latter proof has been mechanized by Nieto and Esparza [30] in Isabelle/HOL [31]. Ben-Ari [8] simplified the DLMSS algorithm but got the informal proof slightly wrong. van de Snepscheut [38] gave a more formal treatment of Ben-Ari's algorithm, corrected some errors, but overlooked others. Chandy and Misra [11] refined a mutator into a mutator/propagator/marker combination using superposition and a version of Ben-Ari's algorithm. Russinoff [36] used an embedding of Manna and Pnueli's temporal logic [27] in Nqthm [9] to verify Ben-Ari's algorithm. We argue that none of these formalizations, including our own prior rendition, satisfactorily specifies a garbage collector. Jones and Yatapanage [25] investigate the use of Rely/Guarantee refinement for the Ben-Ari algorithm with a pre/post-condition form of refinement which is different from the compositional refinement relation we define below since it does not treat the mutator as part of the environment.

In earlier work [20], we had presented a PVS verification of the Ben-Ari algorithm [8] through the use of the refinement mapping technique of Abadi and Lamport [1]. Prior to that we performed a verification of the algorithm without refinement in PVS, and also applied model checking using the Murphi model checker [19]. In this prior work [20], the specification and implementations are presented as transition systems with a state space, an initialization predicate and a transition relation over the state space. Each transition system has an observation function that extracts the observable state space so that we can characterize the observable behavior of the transition system by a trace of the observable behavior of the system where the initialization holds of the initial state and the transition relation holds of each pair of adjacent states. A refinement map between a concrete and abstract transition system maps the concrete states to the abstract states while preserving the observable behavior to show that a concrete initial state can be viewed as an abstract initial state, and given a concrete invariant, the concrete transitions can be simulated by abstract transitions under the refinement map. The concrete invariant plays a key role in enabling the refinement proofs for the garbage collector.

The refinement proof for the Ben-Ari algorithm is carried out in stages. The specification is given as the composition of an abstract mutator and collector, where the mutator redirects an arbitrary node to an accessible (reachable) node, and the collector appends an arbitrary inaccessible node to the free list. The first refinement marks arbitrary nodes and when the accessible nodes have been colored black, the algorithm appends the white nodes to the free list. The second refinement of the collector picks two arbitrary nodes with black-white edges and blackens the target the edge. When there are no more black-white edges, the collector starts appending white nodes into the free list. The third refinement brings the algorithm closer to the Ben-Ari version of the algorithm by scanning memory for black nodes and marking their children nodes, and counting the number of black nodes to establish the absence of a black-white edge before beginning the append phase.

Our prior work had two drawbacks. One, the form of refinement was not compositional. The mutator could be refined to not generate any garbage, and the refinement argument would still go through. Two, the refinement there only covered safety, and could not be used to demonstrate that an unreachable node is eventually added to the free list.

In the present work, we build on the lazy compositional verification approach [37] to define a notion of *compositional refinement* for the verification of the garbage collector where the mutator specification is viewed as a Rely condition for the collector, and the refinement is expected to preserve all possible actions of the abstract mutator. We present a layered compositional refinement of both the DLMSS and Ben-Ari algorithms. The key difference is that we show that the mutator steps are unconstrained by the refinement so that we do have a bisimulation relation on the mutator transitions. We also touch on the preservation of liveness properties in the refinement. Section 2 defines an abstract garbage collector that serves as the top-level specification. Section 3 describes the DLMSS algorithm together with an operational correctness proof followed by a layered compositional refinement. Section 4 offers a similar but abbreviated treatment of the Ben-Ari algorithm followed by concluding observations in Section 5.

An *open transition system* T has the form $\langle \Sigma, \widehat{\Sigma}, I, N, L, \pi \rangle$ with state Σ, observable state $\widehat{\Sigma}$, initialization predicate I, system transition relation N, environment (Rely) transition relation L, and observation function π mapping Σ to $\widehat{\Sigma}$. The behavior $\overline{\sigma}$ of the system is one where $I(\overline{\sigma}_0)$ holds (i.e., I holds of the initial state $\overline{\sigma}_0$, and for each pair of adjacent states $\overline{\sigma}_i, \overline{\sigma}_{i+1}$ in the behavior $\overline{\sigma}$, $N(\overline{\sigma}_i, \overline{\sigma}_{i+1}) \vee L(\overline{\sigma}_i, \overline{\sigma}_{i+1})$ holds. Given an abstract transition system T_1 with invariant P_1 and a concrete transition system T_2 with invariant P_2, the refinement map ρ mapping Σ_2 and Σ_2 must now satisfy the five conditions[4]

1. Observational identity: $\pi(\rho(\sigma)) = \pi(\sigma)$ for $\sigma \in \Sigma_2$.
2. Initiality: $I_2(\sigma) \Rightarrow I_1(\rho(\sigma))$ for $\sigma \in \Sigma_2$.
3. System Simulation: $P_1(\rho(\sigma)) \wedge P_2(\sigma) \wedge P_2(\sigma') \wedge N_2(\sigma, \sigma') \Rightarrow N_1(\rho(\sigma), \rho(\sigma'))$, for $\sigma, \sigma' \in \Sigma_2$.
4. Environment Upward Simulation: $P_1(\rho(\sigma)) \wedge P_2(\sigma) \wedge P_2(\sigma') \wedge L_2(\sigma, \sigma') \Rightarrow L_1(\rho(\sigma), \rho(\sigma'))$, for $\sigma, \sigma' \in \Sigma_2$.
5. Environment Downward Simulation: $P_2(\sigma) \wedge P_1(\rho(\sigma)) \wedge P_1(\widehat{\sigma}') \wedge L_1(\rho(\sigma), \widehat{\sigma}') \Rightarrow \exists \sigma' \cdot \widehat{\sigma}' = \rho(\sigma') \wedge L_2(\sigma, \sigma')$ for $\sigma \in \Sigma_2$ and $\widehat{\sigma}' \in \Sigma_1$.

This ensures that T_2 can work in composition with any environment that satisfies the Rely condition L. In the case of the garbage collector, the Rely condition is the specification of the mutator.

2 Garbage Collection, Abstractly

As a program is executing, it allocates memory nodes to build data structures and redirects pointers from one memory node to another. In this process, some of the allocated memory nodes can become unreachable so that there is no dereferencing path from the program variables to these nodes. A garbage collector collects such unreachable memory nodes and appends them to the free list so that the program (mutator) can reallocate and reuse these previously unreachable nodes. Garbage collection is a popular target for formal modeling and verification. Informal proofs of concurrent garbage collection have been given by Chandy and Misra [11] and by Pavlovic, Pepper, and Smith [34]. Some mechanically verified proofs include a TLA proof [17] of a concurrent garbage collector [15], a safety/liveness proof [22] using an embedding of temporal logic in PVS of the DLMSS algorithm, a formalization [10] of our earlier refinement argument in both B and Coq for the purpose of comparing the two formal systems, and an Isabelle/HOL invariance proof [16] of a tricolor algorithm for the x86-TSO memory model in a multi-mutator setting.

Abstractly, memory M consists of nodes V and directed edges E from nodes to nodes. We restrict ourselves, without loss of generality, to a scenario where nodes only have left edges or right edges, so that $E = E_L \cup E_R$, where E_L (respectively, E_R) contains the left (respectively, right) edges. An edge from source node n to target node n' is represented as $n \xrightarrow{\lambda} n'$, where the label λ is either *left* or *right*. For each node n, there is at most one outgoing left edge or

[4] There are fairness conditions associated with the transitions systems and their refinement, that we omit here.

$\widehat{MU}$	**repeat** $E \ := \ E - \{a \overset{\lambda}{\to} b\} \cup \{a \overset{\lambda}{\to} c\}$ **until** *false* for $a, c \in R$.
$\widehat{C}$	**repeat** $(F, G) \ := \ F \cup \{a\}, G - \{a\}$ **until** *false*, for $a \in G$

Fig. 1. Abstract Mutator and Collector

right edge. For $n \in V$, let $E[n]$ be the set of successor nodes of n, i.e., those nodes n' such that the edge $n \overset{\lambda}{\to} n' \in E$. For $X \subseteq V$, let $E[X]$ be $\bigcup_{x \in X} E[x]$. There is a fixed set of root nodes A. *Reachable* nodes R are those that have a path from a root, namely the nodes in $E^*[A]$. The free list `FREE` is itself such a root node. Each mutator action can *redirect* an edge $a \overset{\lambda}{\to} b$ from any node to a reachable node ($E' = E - \{a \overset{\lambda}{\to} b\} \cup \{a \overset{\lambda}{\to} c\}$ for $c \in R$). In this process, nodes such as b may become unreachable. Thus, the nodes V can be partitioned into the reachable nodes R, which can itself be partitioned into the free nodes $F = E[\texttt{FREE}]$ and the non-free nodes $H = R - F$, and the garbage nodes $G = V - R$. The point of the collector $\widehat{C}$ is to ensure that nodes in G are moved to F. The collector operates concurrently with the mutator $\widehat{MU}$ which might move nodes from R to G. Since the mutator is the program of interest, the collector must do its work without modifying any edges in H.

The abstract system AGC can be in an arbitrary initial state. We can then enumerate the atomic actions of AGC as shown in Figure 1. The transition system is an interleaving of the abstraction mutator action $\widehat{MU}$ and the abstract collector action $\widehat{C}$:

1. The assignment step $\widehat{MU}$ is performed as a single atomic action, and this can affect the sets F, H, and G.
2. The simultaneous assignment $\widehat{C}$ implicitly leaves the set H unchanged. Since F and G are derived features of the state, the abstract collector defines the set of transitions that can modify the memory, i.e., redirect edges, from $M = (V, E)$ to $M' = (V, E')$ so that $F' = E'[\texttt{FREE}] = F \cup \{a\}$ and $G' = V - E'[A] = G - \{a\}$.

The collector action must be fair, i.e., no node is in G indefinitely.

A concrete garbage collector C must refine this abstract collector $\hat{C}$ in the context of its environment $\widehat{MU}$. Since the refinement can create a concrete representation of memory with suitable data structures to assist with garbage collection, the abstract mutator $\widehat{MU}$ is also refined by a concrete mutator MU. Since we want the collector to work with any mutator, the refinement must not constrain the possible mutator actions when going from the abstract to concrete.

3 A Cooperative Garbage Collector

We present the cooperative garbage collector defined by Dijkstra, Lamport, Martin, Scholten, and Steffens [14] (DLMSS). We first present the assumptions and correctness claims for the algorithm and then describe the algorithm followed by an informal operational correctness argument and a more rigorous refinement proof.

3.1 Assumptions and Correctness Claims

There are some simplifications in the DLMSS formulation of the problem that we also adopt:

1. Each node n has exactly two successor nodes $n.\mathit{left}$ and $n.\mathit{right}$.
2. Memory is scannable so that it is possible to iterate over all of the memory nodes. A simple way to formalize such a memory M is an array with indices i ranging from 0 to $|M| - 1$, where $M[i].\mathit{left}$ and $M[i].\mathit{right}$ are the indices of the two successors.
3. `NIL` is a special unmodifiable, self-pointing root node ($E(\texttt{NIL}) = \{\texttt{NIL}\}$). We assume that the memory is abstractly scannable instead of explicitly modeling it as an array.
4. Deletion of an edge is just redirection to `NIL`.
5. Allocation of a node from the free list is also achieved by redirection. Thus, allocation is seen as a mutator step, but there is a sleight of hand in this simplification. One desirable correctness criterion for the collector is that the shape of the memory reachable from non-`FREE` roots can be changed only by mutator actions. This means that the operation of appending a node to the free list should not add or delete any edges that are reachable from non-`FREE` roots.
6. Free list `FREE` is a root node so that all nodes on the free list are in fact reachable.

The correctness criterion as stated informally in the paper is:

1. CC1: Every garbage node is eventually appended to the free list. More precisely, every garbage node present at the beginning of an appending phase will have been appended by the end of the next appending phase.
2. CC2: Appending a garbage node to the free list is the collector's only modification of (the shape of) the data structure.

The correctness criterion does not explicitly assert that only garbage (unreachable nodes) are appended to the free list. This follows from CC2 since if reachable nodes were appended to the free list, the collector would be tampering with reachable edges. Since CC1 and CC2 are stated for the combined mutator/collector systems, there could be trivial ways of satisfying this by, for example, constructing a mutator that never creates inaccessible nodes or that does its own garbage collection. While one could argue that such an implementation is acceptable, this mitigates against the idea that the mutator component should not be constrained by the implementation of the collector.

Mutator	**pick** $a, b \in R \;\cdot\; \langle a.left \;:=\; b; shade(b)\rangle \;[\!]\; \langle a.right \;:=\; b; shade(b)\rangle$
Collector	**repeat** *unmark*; *mark*; *appendToFree* **until** *false*
unmark	$scan\ n \in V \;\cdot\; n.color \;:=\; white$
mark	*markroots*; **repeat** *scanm* **until** $\neg grayfound$
AppendToFree	$scan\ n \in V \;\cdot\;$ **if** $n.color = white$ **then** $append(freelist, n)$
markroots	$scan\ n \in A \;\cdot\; shade(n)$
scanm	$grayfound \;:=\; false$; $scan\ n \in V \;\cdot\;$ **if** $gray(n)$ **then** $shade(n.left)$; $shade(n.right)$; $n.color \;:=\; black$; $grayfound \;:=\; true$

Fig. 2. Concrete DLMSS Garbage Collector

3.2 The DLMSS Algorithm

We describe the steps in (a minor variant) of the DLMSS algorithm and a simplified (compared to the original) version of the proof, and then show that it can be derived as a refinement of the abstract collector together with the mutator specification. An extra *color* field of scalar type {*white*, *black*, *gray*} is added to each node so that *n.color* is either *white*, *black*, or *gray*. A node is said to be marked if its color is black or gray, and unmarked if it is white. A node is shaded if a white node is marked as gray, and is left unchanged, otherwise:

$$shade(n) \stackrel{\triangle}{=} \textbf{if } n.color = white \textbf{ then } n.color \;:=\; gray.$$

The concrete DLMSS algorithm is shown in Figure 2. The collector cycles through three phases: *unmarking*, *marking*, *appending*. In each phase, collector scans the memory to perform a sequence of atomic actions. The scan can be implemented in a number of ways. The simplest is by viewing memory as arranged in an array of $|V|$ nodes from 0 to $|V| - 1$. The other is by allowing the collector some local memory to maintain a set representation of the nodes visited and those that remain to be visited. We treat the scan as a primitive construct of the form $scan\ n \in V \;\cdot\; actions(n)$ to indicate that the nodes $n \in V$ are processed one at a time in some nondeterministic order. The mutator can set either the left or right neighbor of a reachable node a to point to another reachable node b as shown in Figure 2 so that the shading is performed atomically with the redirection. Here **pick** $b \in R \;\cdot\; actions(b)$ represents nondeterministic (demonic) choice of b, $N_1 \;[\!]\; N_2$ represents a nondeterministic (demonic) choice between action N_1 and action N_2, and $\langle N_1, \ldots, N_2\rangle$ represents the atomic execution of a sequence of actions $N_1, \ldots, N_2$. The three phases of the collector can be described as

1. Unmarking: Scan memory to unmark marked nodes as shown in Figure 2. Note that at the end of the unmarking phases, the unreachable nodes are stably (i.e., even with mutator interference) white, and it is stably the case

that there are no black-white (BW) edges both at the end of the unmarking phase and the beginning of the marking phase. This is because the mutator can only shade reachable nodes and cannot create BW edges.

2. Marking: First, the roots are shaded. The algorithm then repeatedly scans the memory to mark all reachable nodes.[5] During the scan, whenever a gray node n is encountered, the child nodes $n.left$ and $n.right$ that are white are colored gray, and the node n itself is colored black. The marking scan operation *scanm* is defined in Figure 2. The Marking phase is terminated when no gray nodes are encountered, i.e., when a scan terminates with $\neg grayfound$. Note that we stably approach the termination of the phase since either *grayfound* is false or the number of white nodes decreases, or the number of white nodes is unchanged but the number of gray nodes decreases. We show below that at the termination of the marking phase, all reachable nodes are stably black, and all nodes in U are stably unmarked.
3. Appending: Scans the marked memory to append all unmarked nodes to the free list.

To summarize, in the DMLSS algorithm, the collector is implemented as follows. Nodes are either unmarked (white), partially marked (gray), or marked (black). A node is shaded when it is colored gray if it was originally white, and left unchanged, otherwise. The collector first unmarks all the nodes. It then marks all the root nodes gray, including `NIL`. Having marked the roots, the collector continues the marking phase by shading all the target nodes b of any gray node a such that $a \xrightarrow{\lambda} b \in E$, and then coloring a black. When the collector's marking scan does not add any new gray edges, the collector collects all white nodes into the free list, and then unmarks all black nodes by coloring them white. The mutator cooperates by shading the node c gray as part of the same atomic action whenever it redirects an edge $a \xrightarrow{\lambda} b$ to an edge $a \xrightarrow{\lambda} c$.

3.3 An Operational Proof of DMLSS

We first present a somewhat operational correctness argument before recasting it as a refinement argument. The proof establishes the invariant that there are no black-white edges during the marking phase of the collector. The collector does not introduce any such black-white edges since it only colors a node black when all its children have been shaded. The mutator also does not introduce any black-white edges since it atomically shades the target node of any redirected edge.[6] The key observation is that if the collector does not encounter any gray nodes during a marking scan, then there must have been no gray nodes at the

[5] The DLMSS algorithm restarts the scan each time a gray node is blackened whereas we complete the scan but do not exit the marking phase until there is a scan where no gray nodes have been encountered.

[6] The DLMSS article [14] discusses the challenge of splitting the mutator redirection operation into separate atomic steps for shading the target node of the redirection and another for the redirection. The proof of the invariant that there are no black-white edges would fail since there is no way to ensure that the target node of a

beginning of the scan, since the collector is the only process that can modify the color of a gray node during a scan, and hence such a node would not be missed by the scan. In other words, any gray nodes at the beginning of the marking scan would be encountered by the collector since the mutator can only color white nodes gray. If there are no gray nodes at the beginning of the scan, then we have a situation where every black node has successor nodes that are shaded, and hence black. Since, at the end of the marking phases, the set of black nodes contain the roots and is closed under successor, they contain the set of reachable nodes. Hence, at the end of the marking phase, any white node must be unreachable, and this property continues to hold during the append phase until the next unmarking phase.

The main invariant is that there are no black-white (BW) edges during the marking phase. We already saw above that a node is only marked black by the collector when all of its children are shaded even with interference from the mutator. The mutator does not create any black nodes. This invariant ensures the properties CC1 and CC2:

1. All white nodes are unreachable when the collector can no longer find a gray node, as already shown above. Thus the append phase attaches only unreachable nodes to the free list.
2. All unreachable nodes are collected within a round consisting of an unmarking, marking, and appending phase. Let U be the set of nodes that are unreachable at the beginning of an unmarking phase, then U is closed under the parent relation since it would otherwise be reachable. The nodes in U are colored white in the subsequent unmarking phase of the collector, and they continue to remain white and unreachable through the next marking phase. This is because, the mutator can only redirect to and mark reachable nodes, and the collector can only mark a node whose parent is marked. Thus all the nodes in U are white and unreachable at the end of the marking phase and hence collected, and none of the edges between nodes in H are redirected by the collector.

3.4 A Compositional Refinement Proof

To recast the above reasoning as a refinement argument, we have to identify the observable and local variables of each component and define a refinement

redirection might be whitened (during the append phase) while the source node is waiting to be blackened. The algorithm also fails, not just the proof: if the mutator shades a node b before redirecting a to b, then collector can complete its marking phase, and then whiten b in the append phase. The mutator can then complete the redirection in the middle of the next marking phase when a happens to be marked black, and b has not yet been shaded. The mutator then deletes all the other edges to b before the collector gets a chance to mark b so that b remains white at the end of the marking phase. In order for the proof above to work, the condition that every black node must have shaded successors must hold prior to each marking scan. The authors introduce a refinement where the mutator when it redirects a from b' to b first shades b' and then redirects a to b, but this algorithm is not of interest here.

map that is used to discharge the refinement proof obligations. The only abstract variable is the memory M consisting of nodes V and maps *left* and *right* defining the edge set E. The concrete memory is also visible and adds a map *color* from V to $\{black, white, gray\}$. The hidden local variables are the *program counter* variables used for tracking the phases and the scans, and the auxiliary variable V that is needed for the fairness proof obligation. The first refinement *DR1* uses the same concrete mutator, i.e., one that atomically shades the target of the redirection, but implements the three phases of the collector as below:

$Unmark_1$	$U \ := \ G$; **repeat pick** $n \in V \ \cdot \ n.color \ := \ white$ **until** $\forall m \in U \cdot white(m)$.
$Mark_1$	**repeat pick** $n \in V - U \ \cdot \ n.color \ := \ black$ **until** $\forall m \in R \cdot black(m)$

Fig. 3. First refinement in DLMSS

1. Unmarking: As shown in Figure 3, this phase is defined to record the unreachable nodes in U and then whiten all the nodes until at least those nodes in U have been whitened. This ensures that the unreachable nodes in U are unmarked at the end of the phase. Since the mutator cannot modify the marking of nodes in U, this phase does terminate with the stable assertion that the nodes in U are unmarked.
2. Marking: The definition in Figure 3 simply blackens arbitrary nodes while avoiding those in U until all reachable nodes have been blackened. This too stably terminates since the number of black nodes increases.
3. Appending: This is also unchanged from the concrete algorithm. Since the appending actions are the only collector actions that are visible, the refinement argument must demonstrate that each appended node is unreachable.

The refinement map between *DR1* and AGC (Figure 1) must be defined to satisfy the following proof obligations:

1. Each *DR1* collector step is either a silent abstract step or is simulated by an abstract collector ($\widehat{C}$) step.
2. The fairness condition on the abstract collector step is met in *DR1* so that any unreachable node is eventually appended to the free list.
3. Each *DR1* mutator step is either a silent abstract step or is simulated by an abstract mutator ($\widehat{MU}$) step.
4. Any abstract mutator step can be simulated by the concrete *DR1* mutator.

The observable concrete memory drops the color field and control states in *DR1*, and is thus identical to the abstract memory so that the definitions of the sets R, F, G, and H are also preserved from abstract to concrete. The obligations are easily discharged.

1. Obligation 1 can be discharged by observing that the set of unappended white nodes is unreachable before and during the **append** phase of the collector, and hence each concrete append step can be simulated by an abstract

append step. The other steps of the collector are simulated by stuttering steps of the abstract collector.

2. For Obligation 2, we have used an auxiliary variable U to record the set G at the start of the unmarking phase. It can then be discharged by showing that
   (a) The nodes in U are all unreachable and white at the end of the subsequent unmarking phase.
   (b) The nodes in U will remain unreachable and white through the marking phase since U is left untouched during the marking.
   (c) Since the nodes in U are white at the end of the marking phase, they are moved to the freelist during the append phase. Thus, any unreachable node is appended to the freelist through at most one complete round of the collector.
3. Obligation 3 can be discharged by showing that the *DR1* mutator action of setting the left or right field of a reachable node to a reachable node, ignoring the shading of the target, is identical to an abstract mutator action. This is the only concrete mutator step so no stuttering steps are needed.
4. Obligation 4 is shown by observing that any redirection operation can be simulated by the *DR1* mutator since the corresponding *DR1* mutator action is always enabled. The only difference is that the *DR1* mutator shades the target node as part of the atomic mutation action.[7]

The next stage of the refinement from *DR1* to *DR2* elaborates the marking phase. We first define the operation $blacken(n)$ and as below

$$blacken(n) \triangleq \begin{array}{l} \textbf{if } n.left.color = gray \wedge n.right.color = gray \\ \textbf{then } n.color \ := \ black. \end{array}$$

With this operation, we can define the marking phase as below.

$$\textbf{repeat pick } n \in V - U \ \cdot \ blacken(n) \ [\!] \ shade(n) \textbf{ until } \forall m \in R \cdot black(m).$$

The refinement map is the identity function on both variables M and U, and the refinement in *DR2* mainly affects Obligation 2. It is easy to see that the *shade* steps are silent whereas the *blacken* steps are directly simulated by the corresponding marking steps in *DR1*. The *DR2* refinement easily yields the invariant that there are no BW edges since we only blacken a node when its children are all stably gray. Recall that such invariants can be employed in the refinement proof obligations.

A further refinement can be done to shade only those nodes that are children of gray nodes, to yield *DR3*. We introduce the *shadechild* operation below to

[7] It is possible that a further refinement of the mutator might involve computation steps other than redirection. These would become stuttering steps at the abstract level, so that the reverse simulation relation would require showing that sequence of concrete steps is always enabled and eventually terminate. Similarly, a refinement of the allocation steps might restrict the nondeterminism of the mutator since the target of the allocation step will be determined by the allocator.

replace the *shade* operation:

$$\mathit{shadechild}(m,n) \triangleq \begin{array}{l} \quad \mathit{shade}(n), \textbf{if } \mathit{gray}(m) \wedge n = m.\mathit{left} \\ [\!] \ \mathit{shade}(n), \textbf{if } \mathit{gray}(m) \wedge n = m.\mathit{right} \end{array}$$

The DR3 marking phase is now defined as

$$\mathit{scanmdr3} \triangleq \begin{array}{l} \quad \textbf{pick } n \in V - U \ \cdot \ \mathit{blacken}(n) \\ [\!] \ \textbf{pick } m, n \in V - U \ \cdot \ \mathit{shadechild}(m,n) \end{array}$$

$$\mathit{markdr3} \triangleq \mathit{markroots}; \textbf{repeat } \mathit{scanmdr3} \textbf{ until } \forall m \in V \cdot \neg \mathit{gray}(m).$$

The refinement from *DR2* to *DR3* follows from the invariant in *DR3* ensuring the absence of BW edges.

The refinement from *DR3* to the fully concrete DLMSS algorithm now becomes easy. The mutator is unchanged as is the appending phase. It is easy to see that the steps in the concrete unmarking scan easily refine those in the unmarking phase of the *DR1*, and hence *DR3*. Specifically, the transition from the unmarking to marking phases is captured by the stable condition that all the nodes in U are unmarked.

The steps in the marking phase of DLMSS can be simulated by *DR3*, including those in *markroots*, since we can ensure that when the concrete collector has shaded all the children of a shaded node, then this condition is stable in the marking phase and hence the step of blackening the gray node does simulate the corresponding step in *DR3*. Finally, the exit condition from the marking phase $\neg \mathit{grayfound}$ can be shown to be equivalent to the check that the reachable nodes have been blackened since we have the invariant that there are no BW edges.

The upshot is that the full specification of the garbage collector is subtle and challenging. The two conditions CC1 and CC2 appear to capture the total correctness, but they ignore the important constraint that the refinement of the mutator/collector combination should not restrict the actions of the mutator. This kind of mixed simulation argument can also be repeated on the mutator side treating the collector as an environment. Any refinement of the mutator that is compatible with the cooperative mutator used for this refinement argument will preserve the correctness properties. For example, we have not given a concrete implementation of the free list, and assumed that it has some graph structure (e.g., a list or a tree) that can be used for appending the collected nodes and for allocating nodes for the mutator. The other interesting challenge is in staging the refinement steps so that the invariants needed for demonstrating the refinement relations are simple to establish at the right level of refinement. The abstract mutator can be seen as a Rely condition on the environment of the collector with the proviso that any refinement of the environment must admit any abstract environment move to be simulated in the concrete.

mutatorb	**pick** $a, b \in R \;\cdot\;$ $a.left \;:=\; b; shade(b)$ $[\!]\; a.right \;:=\; b; shade(b)$
markroots	$scan\ n \in A \;\cdot\; shade(n)$
scanmb	$scan\ n \in V \;\cdot\;$ $if\ black(n)$ **then** $shade(n.left);$ $shade(n.right$
countb	$BC1 \;:=\; \|B\|; BC \;:=\; 0; scan\ n \in V \;\cdot\;$ **if** $black(n)$ **then** $BC \;:=\; BC + 1$
markb	$markroots;$ $BC \;:=\; 0;$ **repeat** $OBC \;:=\; BC$ $BWC \;:=\; \|BW\|$ $BC0 \;:=\; \|B\|$ $scanmb$ $countb$ **until** $BC = OBC$

Fig. 4. Concrete Ben-Ari Garbage Collector

4 The Ben-Ari Algorithm

Ben-Ari [8] modified the tricolor DLMSS algorithm to dispense with gray nodes. In this version, the mutator just marks the target node of a redirection black, and the collector marks any node that is the successor of a black node. In Ben-Ari's variant BA, it is okay for the mutator to decompose the redirect-and-mark step into an atomic redirect step followed by an atomic mark step since we no longer need the invariant ruling out black-white edges. During its marking scan, the collector marks the successors for black nodes. Before each scan, the collector counts the number of black nodes. If this number has increased since the previous count, the scan is repeated.[8]

The concrete Ben-Ari garbage collector is defined in Figure 4. The mutator *mutatorb* now has the atomicity removed, and unlike the DLMSS case, here shading always marks the node black. The unmarking and appending phases are identical to those of DLMSS. The main difference in the marking phase *markb* is that the marking scan alternates with a counting scan in order to determine if the marking has stabilized. As before, the marking phase starts by shading the roots. The marking scan following the shading of the roots is performed by *scanmb*. The counting scan is performed by *countb* which records the initial cardinality of the black nodes in the specification-only variable *BC1* and tallies the black nodes in variable *BC*. Let *BW* represent the set of black-white edges. The entire marking phase *markb* is defined in Figure 4. It repeatedly performs a *scanmb* followed by a *countb* operation until it sees that the current count of black nodes matches that from the prior round. The specification variables

[8] Counting the black nodes is a somewhat crude way to detect termination, but attempts to optimize the termination check are unlikely to work given the atomicity assumptions. Proof attempts of such optimizations along the slines shown here are helpful in revealing the cause of failure.

BWC and $BC0$ record the cardinalities of the the black-white edge set BW and the black nodes, respectively, at the beginning of each round.

4.1 Operational Proof of the Ben-Ari Garbage Collector

Though we lose the invariant on the absence of black-white edges, we can still ensure that when the number of black nodes is unchanged following a scan, then all the white nodes are unreachable. If there are no BW edges, the set of black nodes contains the roots and is closed under successor, and therefore contains the reachable nodes. Thus, if there is a white reachable node, then there must be a BW edge.

The reasoning behind the algorithm is somewhat subtle since we have weakened the atomicity of the mutator action. As before, we present the operational argument before the refinement one. With each scanning step in the marking phase of the collector, either the target of a BW edge has been blackened and the count of black nodes has increased, or the sources of any remaining BW edges have not yet been scanned. With the redirection step of the mutator where node a redirects from b to c, if the target node c is already black, then the count of black nodes remains the same, and we either still have a BW edge with an unscanned source, or a was the source of the only BW edge in which case there are no other remaining BW edges. If the target node c is white, then if it is the target of the chosen BW edge, then clearly the source of this edge (which might be a) has not yet been scanned. If it is not the target of the chosen BW edge, then this edge continues to have an unscanned source. The other mutator action of marking c either increases the count of black nodes or continues to leave a BW edge with an unscanned source.

The result BC returned by the counting scan is a bounded from below by the cardinality $BC1$ of the set of black nodes prior to the scan, and from above by the current cardinality $|B|$. If this count is unchanged from the previous count, i.e., $OBC = BC$, then we can conclude that the cardinality of black nodes remains unchanged during the marking scan. This is because $BC0$ is a lower bound on the count of black nodes before the marking scan, which itself is bounded from below by OBC, and $BC1$ is an upper bound on the count of black nodes after the marking scan. We know that $OBC \leq BC0 \leq BC1 \leq BC$, hence $BC0 = BC1$. However, $BC0 + BWC \leq BC1$, and hence there are no BW edges at the start of the scan. Thus, if the count of black nodes at the end of a scan remains unchanged from the prior count, then there are no reachable white nodes.[9] As with the DLMSS proof, CC1 and CC2 follow since all the white nodes at the end of the marking phase are unreachable, and all the unreachable nodes at the

[9] Russinoff notes that the corresponding argument by Ben-Ari is flawed. It defines a **BW** edge as one from an accessible (i.e., reachable) black node to a white node. If there are three nodes a, b, and c with a pointing to b and b pointing to c, where a and b are blackened, a redirection from ab to ac makes b inaccessible. The fix, given by Russinoff and used here, is to weaken the definition of **BW** so that the source node need not be accessible.

beginning of a collector round are unmarked during the unmarking phase and remain unmarked at the end of the marking phase.

4.2 Compositional Refinement of Ben-Ari Garbage Collector

We now present the compositional refinement steps. The key property of the marking phase can be baked into the refinement as was done in the DLMSS case. The refinement *BR1* is essentially the same as *DR1*. The observable and local variables are the same and the refinement map again just drops the *color* field. The mutator action is as shown above, and the appending and unmarking phases are the same as in *DR1*. As in *DR1*, the marking phase can be simply captured as:

$$\mathit{markbr1} \triangleq \begin{array}{l}\textbf{repeat pick } n \in V - U \ \cdot \ n.\mathit{color} \ := \ \mathit{black} \\ \textbf{until } \forall m \in R \cdot \mathit{black}(m)\end{array}.$$

With this, we know that the marking phase terminates in a stable state where the white nodes are unreachable, and contain the nodes in U. In the next refinement *BR2*, the body of the scan can also be refined to only mark roots or targets of BW edges. This yields

$$\begin{array}{l}\mathit{markroot}(n) \triangleq n.\mathit{color} \ := \ \mathit{black}, \ \textbf{if } n \in A \\ \mathit{markBWtarget}(n) \triangleq \begin{array}{l}\ \ \textbf{pick } m \in V \ \cdot \ \mathit{shade}(n), \textbf{if } \mathit{black}(m) \wedge n = m.\mathit{left} \\ \talloblong\ \textbf{pick } m \in V \ \cdot \ \mathit{shade}(n), \textbf{if } \mathit{black}(m) \wedge n = m.\mathit{right}\end{array}\end{array}$$

In *BR2*, the termination condition $\forall m \in R \cdot \mathit{black}(m)$ can be replaced by the predicate $\forall e \in E \cdot e \notin BW$, asserting that e is not a BW edge. The marking phase *markbr2* can then be defined as

$$\begin{array}{l}\mathit{markbr2} \\ \triangleq \begin{array}{l}\textbf{repeat pick } n \in V \ \cdot \ \mathit{markroot}(n) \ \talloblong \ \mathit{markBWtarget}(n) \\ \textbf{until } \forall e \in E \cdot e \notin BW\end{array}\end{array}.$$

Note that the range of the scan can be relaxed from $V - U$ to V since it can be shown that nodes in U will remain unmarked. The final step to the concrete implementation then consists of replacing the termination check in *BR2* with the counting scan. As a step toward the refinement stage *BR3*, we can define an abstract count *countbr* as

$$\mathit{countbr} \triangleq \mathit{BC1} = |B|; \textbf{pick } (k \mid \mathit{BC1} \leq k \leq |B|) \ \cdot \ \mathit{BC} \ := \ k.$$

The marking phase of *BR3* can then be defined as

```
$\mathit{scanmbr3} \triangleq \mathit{scan}\ n \in V \ \cdot \ \mathit{markBWtarget}(n)$

$\mathit{markbr3} \triangleq \mathit{markroots}; \mathit{BC} \ := \ 0;$
             $\textbf{repeat}$ $\mathit{OBC} \ := \ \mathit{BC};$
                    $\mathit{BC0} \ := \ |B|;$
                    $\mathit{scanmbr3};$
                    $\mathit{countbr}$
             $\textbf{until}$ $\mathit{BC} = \mathit{OBC}$
```

The reasoning behind the refinement between *BR2* and *BR3* is identical to the one in the basic correctness argument above. The marking of the roots with *markroots* can be simulated by *markroot* steps in *markbr2* and the steps in *scanmbr3* can be simulated by the *markBWtarget* steps. The final refinement from *BR3* only needs to show that and *countb* can be simulated by *countbr*, and the termination check $\forall e \in E \cdot e \notin BW$ is equivalent to the check $BC = OBC$.

One interesting difference between DLMSS and BA is the non-atomic sequencing of redirection and coloring. The reason this works here is that the difference between two successive counts is an upper bound on the cardinality of the set of BW edges at the start of the marking phase.

The liveness argument remains largely unchanged from that of DLMSS. A node that is unreachable at the start of an unmarking phase is collected in the following append phase.

5 Conclusions

The main point of the paper is that getting the specification and proof methodology right is an important part of verification. Prior proofs of the garbage collector rely too heavily on operational reasoning so that the claims for correctness only make sense in light of the program itself. For example, we need to examine the program to see that garbage nodes are made accessible through the free list only within the code for the appending phase. With refinement, this is ensured by the proof.

To answer the question in the title of our paper, a garbage collector is a compositional refinement of the abstract garbage collector AGC. Refinement is the right approach to proving the correctness of garbage collectors so that the concrete mutator/collector combination can be shown to implement the abstract garbage collector (AGC) specification. We have contrasted the original operational arguments with the refinement-based approach in order to highlight the elegance of the latter. We have presented an approach to compositional refinement where the actions of the mutator must not be constrained by the refinement. We plan to update our prior PVS formalization of the refinement argument for the Ben-Ari algorithm to use the compositional refinement approach for proving both the safety and the liveness of the garbage collector.

We are both deeply grateful to Cliff for his intellectual leadership and his personal friendship and mentoring. It has been a joy and a blessing to have had the opportunity to interact with him over the decades. So many of the ideas that we routinely use in our work: rely/guarantee reasoning, two-state postconditions, datatype invariants, proof obligations, and retrieve functions, among many others, trace their roots back to Cliff's deep and highly original contributions to our field. VDM [24] (in its different variants) has been tremendously influential as a specification language within the formal methods community and in industry. VDM is one of the key influences for the PVS specification language [33]. Interestingly, modern programming languages increasingly resemble the executable subset of VDM. VDM supports stepwise refinement of an abstract specification

to an implementation, including data refinement (data reification) and operation refinement (operation decomposition). The fundamental data refinement proof obligation involves defining a so-called retrieve function from the concrete data type to the abstract data type, and then showing that concrete operations operate on the concrete data type in a manner compatible with the abstract operations on the abstract data type mapped to by the retrieve function.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Third Annual Symposium on Logic in Computer Science*, pages 165–175. IEEE, Computer Society Press, July 1988.
2. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
3. R. J. R. Back. On correct refinement of programs. *J. Computer and Systems Sciences*, 23(1):49–68, August 1987.
4. Ralph Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
5. Ralph-Johan R Back and F. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(4):513–554, 1988.
6. R.J.R. Back. Refinement calculus, Part II: Parallel and reactive programs. In de Bakker et al. [12], pages 67–93.
7. R.J.R. Back and J. von Wright. Refinement calculus, Part I: Sequential nondeterministic programs. In de Bakker et al. [12], pages 42–66.
8. Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(3):333–344, 1984.
9. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.
10. Lilian Burdy. B vs. Coq to prove a garbage collector. In *the 14th International Conference on Theorem Proving in Higher Order Logics: Supplemental Proceedings*, 2001.
11. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.
12. J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
13. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparrison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Presss, 1998.
14. Edsger W Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth FM Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.
15. Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–83. ACM, 1994.
16. Peter Gammie, Antony L Hosking, and Kai Engelhardt. Relaxing safely: verified on-the-fly garbage collection for x86-tso. In *ACM SIGPLAN Notices*, volume 50, pages 99–109. ACM, 2015.

17. Georges Gonthier. Verifying the safety of a practical concurrent garbage collector. In *International Conference on Computer Aided Verification*, pages 462–465. Springer, 1996.
18. David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, 1977.
19. Klaus Havelund. Mechanical verification of a garbage collector. In José Rolim et al., editor, *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 1258–1283. Springer, 1999.
20. Klaus Havelund and Natarajan Shankar. A refinement proof for a garbage collector. In Ezio Bartocci, Rance Cleaveland, Radu Grosu, and Oleg Sokolsky, editors, *From Reactive Systems to Cyber-Physical Systems - Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday*, volume 11500 of *Lecture Notes in Computer Science*, pages 73–103. Springer, 2019.
21. C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, 1969.
22. Paul B Jackson. Verifying a garbage collection algorithm. In *International Conference on Theorem Proving in Higher Order Logics*, pages 225–244. Springer, 1998.
23. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.
24. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.
25. Cliff B Jones and Nisansala Yatapanage. Investigating the limits of rely/guarantee relations based on a concurrent garbage collector example. *Formal Aspects of Computing*, 31:353–374, 2019.
26. Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Systems Research Center, 130 Lytton Avenue; Palo Alto, California 94301, December 1991.
27. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems—Specification*, volume 1. Springer Verlag, 1992.
28. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, MA, 1965.
29. Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1990.
30. Leonor Prensa Nieto and Javier Esparza. Verifying single and multi-mutator garbage collectors with Owicki-Gries in Isabelle/HOL. In *International Symposium on Mathematical Foundations of Computer Science*, pages 619–628. Springer, 2000.
31. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. Isabelle home page: `http://isabelle.in.tum.de/`.
32. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
33. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995. PVS home page: `http://pvs.csl.sri.com`.
34. Dusko Pavlovic, Peter Pepper, and Douglas R Smith. Formal derivation of concurrent garbage collectors. In *International Conference on Mathematics of Program Construction*, pages 353–376. Springer, 2010.
35. A. Pnueli. The temporal logic of programs. In *Proc. 18th Ann. IEEE Symp. on Foundations of Computer Science*, pages 46–57, 1977.

36. David M Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6(4):359–390, 1994.
37. N. Shankar. Lazy compositional verification. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Revised lectures from International Symposium COMPOS'97)*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, Bad Malente, Germany, September 1997.
38. Jan LA Van de Snepscheut. "Algorithms for on-the-fly garbage collection" revisited. *Information Processing Letters*, 24(4):211–216, 1987.