CGCExplorer: A Semi-Automated Search Procedure for Provably Correct Concurrent Collectors

Martin T. Vechev
Cambridge University

Eran Yahav
IBM Research

David F. Bacon
IBM Research

Noam Rinetzky
Tel Aviv University

Abstract

Concurrent garbage collectors are notoriously hard to design, implement, and verify. We present a framework for the automatic exploration of a space of concurrent mark-and-sweep collectors. In our framework, the designer specifies a set of "building blocks" from which algorithms can be constructed. These blocks reflect the designer's insights about the coordination between the collector and the mutator. Given a set of building blocks, our framework automatically explores a space of algorithms, using model checking with abstraction to verify algorithms in the space.

We capture the intuition behind some common mark-and-sweep algorithms using a set of building blocks. We utilize our framework to automatically explore a space of more than $1,600,000$ algorithms built from these blocks, and derive over 100 correct fine-grained algorithms with various space, synchronization, and precision tradeoffs.

Some tasks are best done by machine,
while others are best done by human insight;
and a properly designed system will find the right balance.
– D. Knuth

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]; D.4.2 [*Storage Management*]: garbage collection

General Terms Verification, Algorithms

Keywords concurrent garbage collection, concurrent algorithms, verification, synthesis

1. Introduction

The design, implementation, and verification of concurrent garbage collection algorithms are important and challenging tasks: As garbage-collected languages like Java and C# become more and more widely used, the long pauses caused by traditional synchronous ("stop the world") collection are unacceptable in many domains. Unfortunately, concurrent collectors are extremely complex and error-prone.

This work attempts to provide a systematic method for the design and verification of concurrent mark and sweep collection algorithms. In our approach, the algorithm designer provides the *insights* behind the algorithm and utilizes our work to *automatically* turn these insights into provably correct collection algorithms. More specifically, the designer specifies a set of "building blocks" that reflect her insights about the coordination between the collector and the mutator (the metadata they share and some constraints on how it may be used). The system then automatically explores the induced space of algorithms, outputting a number of different correct collectors. The correctness of the results is ensured by using specially-designed model-checking with abstraction. When a full coverage of the induced space turns out to be infeasible, we expect the designer to simplify the search.

Interestingly, we found that this form of an iterative design can also benefit the designer: she can start the exploration by specifying only a few building blocks and placing severe constraints; use our framework; study the resulting algorithm; detect a common pattern; and reuse our framework with a more sophisticated input constraining our system to respect the common pattern. Using this form of a staged design, we were able to discover over 100 new (correct) algorithm variations where the only atomicity constraints are at the granularity of a single building block.

Technically, our work builds on the parametric concurrent mark-and-sweep collection framework of [24]. Their framework allowed the derivation of many abstract algorithms by starting from a single simple, very precise, but maximally atomic algorithm instantiation. While the algorithms were simple to derive and prove, they were not directly implementable in an efficient manner. In particular, the mutator write barrier, and the individual marking steps were implemented using large, heavy-weight, atomic sections. Moreover, the time complexity of the algorithms was not linear, their space consumption would exceed that of the heap being collected by a significant constant factor, and their synchronization overhead would be very high.

In this work, we concentrate on transforming a subset of the abstract algorithms from [24] into correct and more lightweight algorithms. Specifically, we transform the large, heavy-weight, atomic sections that form the core of the algorithm of [24] to use a small amount of per-object metadata and perform only a limited amount of computation on each step, resulting in algorithms that are linear in the size of the heap and the amount of mutation.

Our work provides a significant step towards the synthesis of concurrent collectors. We take this step in a modest setting in which the system is comprised of a single collector and a single mutator. We also limit the scope of the synthesis to the mark step of mark-and-sweep algorithms, because this is the part which is most difficult to construct correctly.

The final step in generating an actual implementation would be to parameterize the abstract costs of this model by machine-dependent parameters and the expected application parameters. This information could then be used to automatically select an algorithm for a specific application and to turn the building blocks into machine code. We leave this as future work.

PLDI'07 June 11–13, 2007, San Diego, California, USA.

1.1 Related Work

There has been little work on using formal techniques to automatically discover and verify interesting and practical concurrent algorithms. However, in the field of garbage collection there are several works using formal approaches to verify algorithms.

The works of [8, 9] define a framework to describe generational and conservative collectors. However, it only deals with stop-the-world algorithms. [9] presents a transformational approach using the SETL wide spectrum language to specify an initially correct and inefficient implementation of a stop-the-world collector. Through loop fusion and formal differentiation transformations, they obtain a more precise implementation of a well-known stop-the-world algorithm. In contrast, we derive concurrent collectors.

In [5], a stop-the-world collector is modeled, but not verified using CCS. In [4], separation logic is used to prove the correctness of a stop-the-world copying garbage collector.

Several works formally verify the correctness of Ben-Ari's and Dijkstra's algorithms presented in [3, 10]. Ben-Ari's algorithm has made simplifications to Dijkstra's algorithm with the sole purpose of having an algorithm which is easier to prove. The quadratic complexity of these algorithms is largely due to author's intention of making these algorithms simple and easy to verify.

In [11], Dijkstra's algorithm is verified using the Owicki-Gries logic. A correction to the article was published in [12]. In [18], Ben-Ari's algorithm is verified for both single and multi-mutator systems using Owicki-Gries's logic in the Isabelle/HOL theorem prover [17]. In the work of [14], again Ben-Ari's algorithm is verified using the PVS theorem proving system. Similar work has been done by [19], where he proves Ben-Ari's algorithm but this time in Boyer-Moore's theorem prover. In [15], Dijkstra's algorithm has been verified again in the PVS theorem prover. The paper of [6], proves Ben-Ari's algorithm using the B and Coq systems.

In [22], the authors use model checking with abstraction to verify Dijkstra's and Yuasa's algorithms. In addition, they use a search technique to generate algorithms (and then verify them). However, the algorithm space they consider is rather limited, and the algorithms they derive do not vary in concurrency.

The work of [16] on superoptimization finds the shortest instruction sequence to compute a function. The state space of these algorithms is bounded, while in our problem the state space is unbounded.

In the work of [1], the authors deal with mutual exclusion algorithms. They perform syntactic exploration and discover various interesting algorithms, some of which are better than known solutions under the given space constraints. The authors employ a custom model checker and various heuristics to reduce the state space. However, the state space of these algorithms is bounded a priori, while in our problem the state space is unbounded.

In *sketching* [21], the user provides a reference program of the desired implementation and some sketches which partially specifies certain optimized functions. The sketching compiler automatically fills in the missing low-level details to create an optimized implementation. Sketching has been used to synthesize several bitstream program implementing cryptographic ciphers. In contrast, our work can be seen as a specialized synthesizer applicable to the domain of concurrent mark-and-sweep collectors.

1.2 Main Results

The contributions of this work can be summarized as follows:

- We define a search procedure that explores a space of concurrent garbage collection algorithms built from a set of "building blocks" provided by an algorithm designer.
- We provide a set of building blocks that captures the intuition behind some common mark-and-sweep algorithms.
- We explore the induced space of algorithms and discover over 100 non-trivial solutions.
- We define a model checking (with abstraction) procedure that automatically verifies the correctness of our algorithms.

While the search procedure is general, the abstraction used for verifying the discovered algorithms may be specific to the choice of building blocks. In this paper, we present an abstraction that is appropriate for the subspace of mark-and-sweep algorithms we explore. Due to space restrictions, we only give an informal description of the abstraction. A detailed formal description is given in [23].

Outline. The rest of the paper is organized as follows: Section 2 reviews the framework of [24]; Sections 3-5 gradually present the core of our framework and the new algorithms; Section 6 described the abstraction used for the model-checking algorithm; and Section 7 concludes the paper.

2. The Log-based Parametric Collector

In this section we briefly review the parametric concurrent collection algorithm of [24], which serves as a starting point for our work.

2.1 Log-based Concurrent Collectors

The algorithm is defined using a standard concrete semantics. For simplicity, we assume that all objects have the same set $Fields$ of field identifiers. We denote by *VarId* the set of local variable identifiers. We denote by *locs* an unbounded set of dynamically allocated heap locations, and by $\mathit{Val} = \mathit{locs} \cup \{\mathit{null}\}$ the possible values to which fields and variables may be mapped. A *program state* keeps track of the program locations of the mutator and the collector, a set of allocated objects ($L \subseteq \mathit{locs}$), an environment mapping local variables to values ($\rho \colon \mathit{VarId} \rightarrow \mathit{Val}$), and a mapping from fields of allocated objects to values ($h \colon \mathit{locs} \rightarrow Fields \rightarrow \mathit{Val}$). For convenience, we use $obj.f$ to denote the value $h(obj)(f)$.

All reachable objects are reachable from a finite set of $R \subseteq L$ root objects, denoted $root_1, \ldots, root_R$. No particular structure is assumed regarding the initial root set for collection. Thus it could consist of just a single pointer, the stack of one thread, or the stacks of multiple threads. (For simplicity, stack frames are treated as heap allocated objects.)

The collection algorithm uses an *interaction log* to record information about the combined behavior of the collector and the mutator. This log is used by the collection algorithm to select the objects to be marked.

The interaction log is a sequence of log entries of the following kinds: (i) a *tracing* entry recording a tracing action of the collector as it traverses the heap during the marking phase; (ii) a *mutation* entry recording a pointer redirection action by the mutator; (iii) an *allocation* entry recording an allocation of a new object by the mutator. This is formally defined as follows:

DEFINITION 2.1. *A* log entry *is a tuple* $\langle \mathbf{k}, source, fld, old, new \rangle \in \{\mathbf{T}, \mathbf{M}, \mathbf{A}\} \times L \times Fields \times \mathrm{Val} \times \mathrm{Val}$*, where:*

- $\mathbf{k}$ *identifies the kind of action as one of tracing, mutation, or allocation, denoted by* **T, M,** *and* **A***, respectively.*
- $source$ *is the object affected by the action.*
- fld *is the field of* $source$ *affected by the action.*
- old *is the value of the field* $source.fld$ *prior to the action.*
- new *is the value of* $source.fld$ *subsequent to the action.*

Tracing actions do not change the structure of the heap; therefore $old = new$ for all tracing entries. Allocation actions allocate the object new, which must not appear previously in the trace.

The following selectors are used to access the components of a given log entry tuple, $\tau = \langle \mathbf{k}, s, f, o, n \rangle$: $\tau.kind = \mathbf{k}$, $\tau.source = s$, $\tau.fld = f$, $\tau.old = o$, and $\tau.new = n$.

```
collect() {
  atomic
  [ marked ← {root_1, ..., root_R}
  [ pending ← ⋃_{x∈marked} fields(x)
  [ log ← ε

  do {
    mark()
    addOrigins()
  } while (?)

  atomic
  [ addOrigins()
  [ mark()

  sweep()
}
```

(SKL) Collector skeleton

```
mark() {
  while (pending ≠ ∅) {
   (obj, fld) ← removeElement(pending)
   atomic
   [ dst ← obj.fld
   [ log ← log • ⟨T, obj, fld, dst, dst⟩

   if (dst ≠ null ∧ dst ∉ marked){
      marked ← ∪ {dst}
      pending ← ∪ fields(dst)
}}}
```

(MRK) Collector tracing

```
addOrigins() {
  atomic
  [ origins ← expose(log) \ marked
  marked ← ∪ origins
  pending ← ∪ (⋃_{x∈origins} fields(x))
}
```

(ORG) Collector adding origins

```
mutate(source, fld, new) {
  atomic
  [ old ← source.fld
  [ log ← log • ⟨M, source, fld, old, new⟩
  [ source.fld ← new
}

mutateAlloc(source, fld) {
  atomic
  [ new ← alloc new object
  [ old ← source.fld
  [ log ← log • ⟨A, source, fld, old, new⟩
  [ source.fld ← new
}
```

(MUT) Mutator write and allocation barriers

Figure 1. A log-based parametric Mark-and-Sweep algorithm.

2.2 The Parametric Algorithm

Fig. 1 presents the pseudo-code for a parametric concurrent mark-and-sweep collector. The operation of this collector is defined over a prefix of the interaction log, recording the collector and mutator interaction. (In the figure, we use $log \bullet \tau$ to denote a concatenation of τ at the end of log.)

The parametric collection algorithm does not specify how objects are selected to be marked. Instead, the parameter function *expose* determines how the collector handles objects that may be "hidden" due to concurrent mutations. The algorithm, however, does restrict concurrency by assuming that write barriers are atomic with respect to collector operations. Effectively, this means that a collector cannot preempt a mutator during a write barrier.

The collection cycle of the algorithm is described in the `collect` procedure. The collection cycle consists of two phases: (i) the *marking phase*, in which the collector marks potentially live objects; (ii) the *sweeping phase*, in which unmarked objects are reclaimed.

The collection cycle starts by atomically selecting the set of root objects as origins. After selecting the root objects as origins, the collector proceeds by repeatedly tracing heap objects and marking them (`mark` procedure), and adding origins to be considered by the collector due to concurrent mutations performed by the mutator (`addOrigins` procedure). These two steps are repeated until a non-deterministic choice (denoted by '?' in the figure) triggers a move to an atomic phase in which the remaining origins and objects to be marked are processed atomically. This atomic phase guarantees the termination of the algorithm, and is in line with some practical collector implementations (e.g., [2]).

After the marking phase has completed, the sweep phase reclaims all objects that are not marked. Neither [24] nor we specify how the `sweep` operation proceeds, except to ensure that there is the proper synchronization between the mark and sweep phases.

Marking Traversal: The `mark` procedure implements a collector traversal of the heap. (obj, fld) denotes the field fld of an object obj. $fields(obj)$ denotes the set of all object fields for a given object obj. The procedure uses a set $pending$ of pending fields to be traversed, and performs a transitive traversal of the heap by iteratively removing an object field from $pending$ and tracing from it. Whenever an object field is traced-from, the procedure inserts a tracing entry into the log. When the traced object field points to an unmarked object, the object is marked, and its fields are added to the pending set.

During this traversal, the mutator might concurrently modify the heap. These concurrent mutations might cause reachable objects to be *hidden* from the traversal, and thus may remain unmarked by the *current* traversal.

The Collector Wavefront: All collectors discussed in this paper rely on cooperation between the collector and the mutator to guarantee correctness in the presence of concurrency. A key part of the cooperation is tracking the progress of the collector through the heap, since mutations can be treated differently depending on whether they happened in the portion of the heap already scanned by the collector (behind the wavefront) or not yet scanned (ahead of the wavefront). The progress of the collector is tracked by tracking the set of object fields (that is, *not* the values of the pointers in those fields) that have been traced by the collector thus far.

DEFINITION 2.2. *Given a log prefix P, the set of object-fields that have been traced by collector operations in P is:* $\mathcal{W}(P) = \{(P_i.source, P_i.fld) \mid P_i.kind = \mathbf{T} \wedge 0 \leq i < |P|\}$*, where P_i denotes the ith log entry in P. We say that an object field (o, f) is* behind the wavefront *when $(o, f) \in \mathcal{W}(P)$, and* ahead of the wavefront *when $(o, f) \notin \mathcal{W}(P)$.*

Adding Origins: The `addOrigins` procedure uses the interaction log to select a set of additional objects to be considered as origins. When this procedure is invoked by the collector, it is possible that a number of reachable pointers were hidden by the mutator behind the wavefront during the `mark` procedure. The `addOrigins` procedure finds a safe over-approximation of these hidden, but reachable objects.

The core of `addOrigins` is the *atomic* call to $expose$. The latter takes a log prefix and returns a set of objects that should be considered as additional origins. Each object returned by $expose$ is then marked, and its fields are inserted into the $pending$ set.

Mutator Barriers: Fig. 1(MUT) shows the write-barrier and allocation-barrier used by the mutator. The procedure `mutate` is called by the mutator to update a pointer in the heap. The procedure `mutateAlloc` is called by the mutator to allocate a new object and store it in the given field. To collaborate with the collector, the mutator barriers append their actions to the interaction log.

When the mutator performs an assignment $source.fld \leftarrow new$ with $new \neq \textbf{null}$, we say that a pointer is *installed* from $(source, fld)$ to new. When the object field $(source, fld)$ is behind the wavefront, we say that the pointer is *installed behind the wavefront*. Otherwise, we say that the pointer is installed ahead of the wavefront. Similarly, whenever we assign a value to a field

$(source, fld)$ containing an existing pointer, we say that the existing pointer is *deleted*. If the field $(source, fld)$ is ahead (behind) of the wavefront, we say that the pointer is *deleted ahead (behind) of the wavefront*.

Counting-Based Collection: We now describe an approach for exposing hidden objects, which is based on counting the number of references to an object from behind the wavefront.

The mutator count is the number of pointers to an object from object fields behind the wavefront. This quantity is computed with respect to a given wavefront. Note that this is *not a general form of reference counting*.

To compute the mutator count from a given log prefix P, we define the mutator-count increment and decrement as follows ($\mathrm{pre}(P, i)$ denotes P's prefix of length i):

$$M^{+}(o, P) = |\{P_i \mid P_i.kind \in \{\mathbf{M}, \mathbf{A}\} \wedge P_i.new = o \\ \wedge (P_i.source, P_i.fld) \in \mathcal{W}(\mathrm{pre}(P, i)) \wedge 0 \leq i < |P|\}|$$

$$M^{-}(o, P) = |\{P_i \mid P_i.kind \in \{\mathbf{M}, \mathbf{A}\} \wedge P_i.old = o \\ \wedge (P_i.source, P_i.fld) \in \mathcal{W}(\mathrm{pre}(P, i)) \wedge 0 \leq i < |P|\}|$$

The value $M^{+}(o, P)$ is the number of new references introduced by the mutator from object fields that are behind the wavefront. Similarly, the value $M^{-}(o, P)$ is the number of references removed by the mutator from object fields behind of the wavefront. The mutator count $M(o, P)$ is computed by combining the mutator-count increments and decrements as follows:

$$M(o, P) = M^{+}(o, P) - M^{-}(o, P)$$

A counting-based collector can be instantiated using the following $expose^{c}$ function.

$$expose^{c}(P) = \{n \mid n = P_i.new \wedge M(n, P) > 0 \wedge 0 \leq i < |P|\}$$

In real systems, the count maintained for an object is usually very small. Therefore, it would be wasteful to have a very large reference count per object. We therefore introduce a *threshold* for the count. The threshold limits the mutator count to a maximum value, at which it "sticks" and is not subsequently decremented. This allows counts to be implemented with a fixed (small) number of bits while still maintaining the correctness properties provided by reference counting.

3. From Log-based to Log-free Algorithms

In this section, we perform several manual steps that allow us to move away from the log-centric model of our algorithms and move towards a more practical model that allows some of the collector-related computation to take place in the write barrier.

Specifically, we show how to get a collection algorithm that does not use a log by retaining the collector skeleton, shown in Fig. 1(SKL), and fixing certain design decisions regarding the implementation of `mark`, `addOrigins` and `mutate`. Practically, this means adding additional metadata (state) per object. Note that Fig. 1 is exactly the same algorithm discussed in [24] and is repeated here for convenience.

The algorithm obtained in this section is shown in Fig. 2. In this figure, parts (MRK), (ORG) and (MUT) of Fig. 1 are modified from being log based to being state based, as will be shown below.

The collector skeleton of Fig. 1(SKL) is identical for both algorithms and is not shown in Fig. 2. The code for (MRK), (ORG), and (MUT) shown in Fig. 1 is converted to the code shown in the corresponding parts of Fig. 2.

For clarity of presentation we do not show the necessary range checks for an overflow on increment (operation $new.MC++$). One can assume the operations $new.MC++$ and $new.MC--$ are no longer activated if overflow of the count has occurred. Similarly, we do not show that all operations in the barrier but the actual field assignment are predicated on their target object being non-null and unmarked.

The following manual steps are taken to perform the conversion.

Update the Collector Wavefront On-the-fly: Rather than recomputing the wavefront each time it is used by $expose$, the collector keeps track of the wavefront and updates it incrementally in the `mark` procedure. Technically, the incremental wavefront can be implemented by storing a designated bit on each field recording whether the field has been scanned by the collector. We denote the wavefront bit for a field fld by WF_{fld}.

Introduce a Marked Bit: Instead of recording the marked objects in a collector data-structure, we use a bit on every object to record whether the object has been marked by the collector. We use the field $mark$ to hold the mark bit of an object.

Constrain Object Processing Order: In the log-based algorithm, the fields of an object are processed in an arbitrary order (i.e. $pendingFields$). There are two sources of non-determinism here. First, it is possible to process fields of different objects in arbitrary order and second, it is possible to process fields of the same object in arbitrary order.

This step eliminates both sources of non-determinism by processing all fields of an object in a predefined order. This allows us to maintain a pending set of objects, rather than a pending set of fields, and hence use less space. (In Fig. 2, we use the variable $pending$ instead of $pendingFields$ to denote the set of pending objects.) Technically, we assume that fields within an object are identified by a number and are processed in an increasing order.

Moving Computation from the Collector to the Mutator: In this paper, we will concentrate solely on the counting-based algorithms from [24] which allocate black objects. To avoid re-computation of the mutator count, we introduce a counter MC for each object. This counter can be stored in the object header. The MC counter is updated incrementally by the mutator's write-barrier.

Because both wavefront and count information are stored in heap objects and updated incrementally, the log used in the log-based algorithm of Fig. 1 is reduced to a set of objects for which the count should be re-inspected when the tracing is over (at the point of expose). We denote this set of objects by $cand$, as these objects are candidates for being exposed.

Next, the $expose$ function is adapted as well. We denote the adaptation of the expose function that only reinspects the object counts by $\widehat{expose}$. The adapted expose function is defined as follows: $\widehat{expose}(cand) = \{o \mid o.MC > 0 \wedge o \in cand\}$

So far, we have only defined $\widehat{expose}$ as a function over the candidate set $cand$. However, since we are interested in exploring versions of $\widehat{expose}$ that are not fully-atomic, we need to describe it operationally. We choose to represent the candidate set $cand$ using an array, and the $\widehat{expose}$ operation via a traversal of the array (atomically at this point, but we relax atomicity constraints in later sections). The variable $nextcand$ represents the current size of the candidate set. It is updated by the mutator when it inserts references into $cand$ and is read by `addOrigins` when it processes the $cand$ set. In our experiments we have simplified the problem by assuming that the set $cand$ only grows. In a more practical implementation the collector would need to update $nextcand$ when it finishes processing the set, e.g. by resetting it to zero.

Separate Shared Computation from Local Computation: Initially `mark` and `addOrigins` consisted of updates to both heap and local information (e.g. $pendingFields$ is a variable local to the collector). We would like to separate these two because we are interested in concentrating and varying only the part which is difficult to create and prove correct: the code accessing the shared data.

Fig. 2 shows the routines performing collector-local computation: `mark`, and `processObject` in the (MRK) part, and

```
mark() {
  while (pending ≠ ∅) {
    obj ←removeElement(pending)
    processObject(obj)
}}

processObject(obj) {
  for (fld = 0; fld < |Fields|; fld++) {
    dst ← markStep(obj,fld)
    if (dst ≠ null)
      pending ← ∪ {dst}
}}

markStep(obj,fld) {
  atomic
    [ dst ← obj.fld
    [ obj.WF_fld ← true
    [ if (dst ≠ null ∧ ¬dst.mark)
    [       dst.mark ← true
    [ else dst ← null
  return dst
}
```

(MRK) Collector tracing

```
addOrigins() {
  atomic
    [ for (i = 0; i < |nextcand|; i++){
    [   dst ← addOriginStep(i)
    [   if(dst ≠ null)
    [     pending ← ∪ {dst}
    [ }
}

addOriginStep(i) {
  dst ← cand[i]
  dst.inLog ← false
  if (dst.MC > 0 ∧ ¬dst.mark)
    dst.mark ← true
  else dst ← null
  return dst
}
```

(ORG) Collector adding origins

```
mutate(source, fld, new) {
  atomic
    [ if(source.WF_fld){
    [   old ← source.fld
    [   new.MC++
    [   old.MC−−
    [   if(¬new.inLog){
    [     cand[nextcand++] = new
    [     new.inLog ← true
    [ }}
    [ source.fld ← new
}

mutateAlloc(source, fld) {
  new ← alloc new object
  mutate(source, fld, new)
}
```

(MUT) Mutator barriers

Figure 2. Atomic log-free mark-and-sweep collector with metadata stored on heap objects and mutator-computation of the mutator count.

`addOrigins` in the (ORG) part. These procedure do not directly read and write to heap objects, and use the procedures `markStep` and `addOriginStep` to access shared heap data.

Part (MUT) of the figure shows the mutator routines: the write barrier procedure `mutate` that accesses shared data and the allocation barrier `mutateAlloc`.

In the figure, we show the procedures that mutate shared heap data with a shaded background. These are the procedures that will be automatically synthesized by our framework.

It is worth noting that `addOrigins` and `mutate` are accessing a shared variable $nextcand$. We will revisit this point in Section 5.5.

Although the procedure `addOrigins` is atomic, we chose to show the splitting of `addOrigins` and `addOriginStep` here as in Section 5, we will be concentrating on removing atomicity for both `addOrigins` and `addOriginStep`.

To summarize, the resulting algorithm has the following characteristics:

- **space overhead**: the algorithm uses: (i) one bit per object-field in order to record the collector's progress through the heap (wavefront); (ii) one $mark$ bit per object; (iii) one $inLog$ bit per object recording whether the object is in the candidate set; (iv) a counter per object whose number of bits depends on the counting threshold.
- **concurrency**: the algorithm uses an atomic write-barrier, and two atomic collector steps: `markStep` and `addOrigins`. In effect, the collector cannot perform a marking step or `addOrigins` while the write barrier is executing, and the mutator cannot mutate the heap in the duration of a tracing step (tracing a single field) or during `addOrigins`.
- **precision**: this algorithm has the same precision as the log-based algorithm. All algorithms we derive in this work are of identical or lesser precision than the algorithm we obtained in this section.

In this section we obtained an algorithm that uses large coarse-grained atomic routines: `markStep`, `mutate` and `addOrigins`.

The key question that we address in the rest of the paper is: what are the (correct) alternative implementations in terms of concurrency to the three coarse-grained routines described above? Can we obtain other less-atomic implementations of these three routines? Can we find implementations that use no atomics at all? This is a challenging question because the collector and the mutator concurrently manipulate shared metadata.

We approach this question *systematically*, and, using our automated exploration procedure, find a number of interesting correct collection algorithms.

4. The Exploration Framework

In this section, we describe our semi-automated framework for exploring a space of collection algorithms. In Section 5, we put the framework to work, and derive a variety of collection algorithms by systematically breaking the coarse-grained atomicity of the algorithm described in Section 3.

Our framework is composed of a search algorithm and a verification procedure. The search algorithm attempts to verify the correctness of algorithms in the space as they are being explored. In this section, we describe the search algorithm, and treat the verification procedure as a black box (the details of the verification procedure are described in Section 6).

Our search procedure looks for correct implementations of three procedures: `markStep`, `addOriginStep` and `mutate`.

4.1 Atomicity

We use the key word *atomic* to denote that a sequence of operations should not be interrupted.

We will be using the terms *more atomic*, *most atomic* and *less atomic* with respect to collection algorithms. We define these relations in terms of interleavings. We define an interleaving of an algorithm in the standard way: a sequence of collector and mutator operations, where each operation is an execution of a *building block* as defined in the next subsection. We denote the set of interleavings for an algorithm C_1 by $int(C_1)$. We note that only algorithms comprised of the same building blocks and in the same order (but, possibly, with different atomicity constraints) are comparable by this partial order. Therefore, given two collector algorithms C_1 and C_2, we say that C_1 is *more atomic* than C_2, when $int(C_1) \subset int(C_2)$. Subsequently, given a set of collection algorithms S, $c \in S$ is the most atomic if for any other $c' \in S$, $c' \neq c$, and c is more atomic than c'. The definitions for *less atomic* and *least atomic* are similar.

Collector		
BB#	**Building Block**	**Meaning**
$C1$	$dst \leftarrow o.fld$	scan field
$C2$	**if** $(dst \neq null \wedge \neg dst.mark)$ $dst.mark \leftarrow true$ **else** $dst \leftarrow null$	mark an object which is not already marked
$C3$	$addOrigins()$	process candidate objects
$C4$	$o.WF_{fld} \leftarrow true$	notify field is read

Mutator		
BB#	**Building Block**	**Meaning**
$M1$	$old \leftarrow o.fld$	read field
$M2$	$o.fld \leftarrow new$	write field
$M3$	$if(val)\ o.\texttt{MC} \leftarrow o.\texttt{MC} + 1$	conditional inc of count
$M4$	$if(val)\ o.\texttt{MC} \leftarrow o.\texttt{MC} - 1$	conditional dec of count
$M5$	$cand \leftarrow cand \cup \{o\}$	records object as candidate
$M6$	$val \leftarrow o.WF_{fld}$	check if field is read

Table 1. Building blocks forming the algorithm of Fig. 2.

```
set explore(Blocks,Constraints) {
 correctSet = ∅;
 algSpace = genAll(Blocks,Constraints);

 while (algSpace ≠ ∅) {
  alg = pickandRemoveMostAtomic(algSpace);
  if (verify(alg)) {
   // propagate correctness
   correctSet = correctSet ∪ moreAtomic(alg,algSpace);
   algSpace = algSpace \ moreAtomic(alg,algSpace);
   correctSet = correctSet ∪ {alg};
  } else {
   // propagate failure
   algSpace = algSpace \ {alg};
   algSpace = algSpace \ lessAtomic(alg,algSpace);
 }}
 return correctSet;
}
```

Figure 3. Exploration Procedure.

4.2 Input

The input of the exploration framework is the designer specification which is comprised of a set of *building blocks* and, optionally, some *constraints* over them.

Building Blocks: The building blocks can be viewed as the statements of a domain specific language for constructing concurrent collectors. In our framework, the building blocks have the property that they can be implemented via a small number of machine instructions. Generally, any building block that references storage, where the storage may be accessed concurrently (due mutator-collector concurrency) contains at most two memory accesses (although it may operate upon any number of local values). However, it is possible to have building blocks which consist of higher numbers of memory accesses (generally, enclosed in atomic sections.)

For example, Table 1 shows a choice of building blocks that make up the coarse-grained atomic algorithm of Fig. 2. Initially, we start by treating the updating of the $cand$ set as an atomic operation (in `mutate`). We also treat the collector's `addOrigins` as atomic. Therefore, these two steps are represented as single atomic building blocks, respectively $M5$ and $C3$, in Table 1. In Section 5, we will reduce the atomicity of `addOrigins` further.

We denote the set of all specified building blocks by $Blocks = CBlocks \cup MBlocks$, where $CBlocks$ and $MBlocks$ denote the (disjoint sets of) building blocks performed by the collector and the mutator, respectively.

Constraints: Our framework supports two forms of (optional) constraints: *ordering constraints* and *atomicity constraints*.

The collector ordering constraints require that the collector's blocks are executed in a certain order. They should form a partial order over the building blocks. The same property holds for mutator ordering constraints.

The atomicity constraints specify which blocks should be executed atomically. They should form an equivalence relation over the building blocks.

Both forms of constraints do not allow building blocks from the collector and the mutator to occur in the same constraint. That is, we cannot have an ordering constraint which specifies that $C2$ occurs after $M1$, but we can have an ordering constraint that specifies that $C2$ occurs after $C1$. Providing constraints serves several purposes. First, without some constraints, exploration my be infeasible as there may be too many points in the space to verify. Second, even if it is feasible, it is still possible to reduces the size of the algorithm space and thus makes the exploration more efficient. Third, constraints may be a natural way for the designer to express insights that he is aware of.

4.3 Exploration

Fig. 3 shows the algorithm we use for exploring the algorithm space. `explore` uses the designer's specification to automatically and systematically explore the space of collection algorithms that can be defined using the input blocks and constraints.

Algorithm Generation: The `explore` procedure starts by invoking `genAll` which initializes the $algSpace$ set of algorithms for exploration by enumerating all combinations of building blocks and atomicity constraints that satisfy the input constraints.

`genAll` generates all the algorithms in the space using the following two operations: (i) sequential composition, i.e., placing an ordering constraint on building blocks within the collector (and respectively within the mutator); (ii) combining operations into atomic sequences (effectively melding building blocks together). For efficiency, `genAll` performs a lazy enumeration and not a single eager enumeration step. This is more efficient because it saves storage and improves the search. That is, as the search proceeds, it is possible to infer that certain algorithms are correct without needing to enumerate them. For example, an algorithm A is correct if a less atomic version B is also correct, and hence we can infer B's correctness without needing to explicitly enumerate it.

Exploration: The `explore` procedure iteratively processes the set $algSpace$ and tries to verify each algorithm by calling a model checker (procedure `verify`). When verification succeeds for an algorithm alg, it is added to the set of correct algorithms, and its correctness is used to imply the correctness of all algorithms that are more atomic than alg. This is done by adding the algorithms that are more atomic than alg to the set of correct algorithms, and removing them from the $algSpace$ set, as they are guaranteed to be correct and need not be considered further. If the verification of alg fails, `explore` removes all less atomic variants of alg from the $algSpace$ set, as they are guaranteed to be incorrect.

The procedure `pickandRemoveMostAtomic` is a particular heuristic choice that we made in our exploration. The exploration always selects the most constrained algorithm in terms of atomicity from the $algSpace$ set, and tries to find correct algorithms that are less constrained. Effectively, we are expecting most variations to be incorrect and expect to propagate the incorrectness to less atomic variations. It is certainly possible to have a hybrid approach where we start from the least atomic algorithm and expect it to be correct so that we propagate that information to the more atomic variations. At any rate, such techniques could be used to reduce the number of algorithms that need to be checked by the model checker and hence explore larger algorithm spaces. Additionally, there is an opportunity for running `explore` on several processors.

Note that the search algorithm is parametric in the set of building blocks. That is, we can add additional building blocks and automatically explore a new algorithm space. However, the verification procedure is not parametric and would need to be adapted.

Optimizations: As mentioned earlier, we assume that the fields of an object are processed in increasing order. Experimentally, the utilization of this assumption reduces the state-space and speeds up the checking of the algorithm. Moreover, the propagation of correctness and failure reduce the number of calls to the model checker by more than 90%, which is important because the `verify` procedure dominates the total exploration time.

To limit the search space, the exploration system checks for algorithms where every building block is used once. If a designer wishes a block to be used more than once, she would have to specify that as a separate block.

For efficiency, we also equate algorithms that differ only in the order of (non data dependant) building blocks that are executed atomically, that is $atomic$ $[a\ b]$ is equivalent to $atomic$ $[b\ a]$ if building blocks a and b are not data dependent.

4.4 Output

The exploration procedure returns a set of correct concurrent collectors. Every algorithm is a pair of a write-barrier description and a collector-step description. The collector-step consists of the procedures `markStep` and `addOriginStep`. In cases where we consider `addOrigins` to be atomic, the collector step will be simply consist of the `markStep`.

The algorithms are encoded using a *symbolic representation*. A symbolic write-barrier description consists of: a sequence of mutator blocks and some atomicity constraints over them (i.e., an equivalence relation). A symbolic collector-step description is similar, but uses collector blocks.

The order of blocks in the sequence reflects the (total) order in which these blocks are executed. The atomicity constraints specifies which blocks should be executed atomically. These constraints refine the designer's specification. The algorithm also respects the data dependencies between building blocks and does not explore orders in which building blocks use uninitialized values.

Given two blocks $B_1, B_2 \in Blocks$ and a sequence seq, we write $B_1 < B_2$ when B_1 precedes B_2 in seq. Note this forms a partial order.

Given two blocks $B_1, B_2 \in Blocks$ and a set of atomicity constraints $acs \subseteq Blocks \times Blocks$, we write $[B_1, B_2]$ when $(B_1, B_2) \in acs$, denoting the fact that acs requires that B_1 and B_2 are executed atomically. Note that this is an equivalence relation.

5. Discovering Algorithms

In this section we describe how we used the automated system to synthesize a number of interesting fine-grained synchronization algorithms. We show several possible scenarios that we experimented with and the results that were obtained from each.

This section demonstrates the process that a typical user of the system would follow in order to synthesize algorithms. In this example the system is used in a staged manner, utilizing the feedback of the system over a particular set of building blocks in order to reduce the search space for a more refined set of building blocks. In that sense, we derived variations of algorithms incrementally. As we progress through this section, we describe how we used the system to explore various sets of building blocks. We summarize the results presented in this section in Table 2. The last four columns of the table show the number of algorithms explored at each step (Total), the number of algorithms model-checked by the checking procedure (SPIN), the number of correct algorithms found (Correct), and the exploration time (in minutes). The difference between the columns Total and SPIN is that the correctness or failure of many algorithms (from Total) can be deduced without invoking the SPIN checker (as discussed in the previous section). It should be noted that the Correct column indicates the number of least atomic correct algorithms. More atomic algorithms are trivially correct and their number is not shown in this column. As noted previously, the running times are affected by our particular choice of exploration. Different choices such as starting from the least atomic algorithm or a hybrid approach will yield different running times. In our experiments we checked the safety property of the algorithms, i.e., that all reachable objects are marked and processed at the end of marking. Further details of the verification are available in Section 6.

Sec.	Run	Total	SPIN	Correct	Time (min.)
5.1	(a)	306	45	1	2
5.3	(b)	2744	162	2	34
5.4	(c)	12	7	2	1
	(d)	592	146	14	56
	(e)	32	26	1	1
5.5	(f)	3024	550	80	212
	(g)	-	-	-	T/O
	(h)	6144	127	10	39
	(i)	1,624,320	1833	6	2072
	(j)	364,032	288	0	39

Table 2. Results of Exploration. Experiments performed on a machine with a 3.8 Ghz Xeon processor and 8 Gb memory running version 4 of the RedHat Linux operating system.

$mutate_4()$
$old \leftarrow o.fld$
detectedAtomic
$o.fld \leftarrow new$
$w \leftarrow o.WF_{fld}$
atomic
$if(w)\ new.MC{+}{+}$
$if(w)\ cand \leftarrow cand \cup \{new\}$
$if(w)\ old.MC{-}{-}$

$markStep_4()$
detectedAtomic
atomic
$dst \leftarrow o.fld$
if $(dst \neq null \wedge \neg dst.mark)$
$dst.mark \leftarrow true$
else $dst \leftarrow null$
$o.WF_{fld} \leftarrow true$
return dst

Figure 4. The least atomic algorithm that is automatically detected by our system from the building blocks of Table 1 (depicted as run (a) in Table 2).

As users of the system, we start the process with some intuition as to what are reasonable building blocks from which an algorithm could be constructed, but without knowing the full details of how to put these blocks together. This process is similar to the idea of sketching. Different users of the system may come up with different such insights/building blocks.

As mentioned in Section 3, the algorithm skeleton has already been split into local and shared parts manually as shown in Fig. 2. In what follows, we will describe how we used to system to fill the shared routines (shaded in Fig. 2) with correct code.

5.1 Exploration: Starting Point

The starting point of our search is the coarse-grained algorithm of Fig. 2. There are two main causes of mutator-collector interference in this algorithm. The first is the interaction between `markStep` and `mutate`. The second is the interaction between `addOrigins` and `mutate`.

As a first step, we decide to try and resolve the interference between `markStep` and `mutate`, and leave `addOrigins` atomic. That is, `addOrigins` is a single building block. Note that although no operation in `mutate` can preempt an atomic `addOrigins`, making `mutate` non-atomic will allow for it to be preempted by `addOrigins`.

The building blocks comprising the shaded parts of the algorithm of Fig. 2 were already shown in Table 1. Our first natural choice is to run the system with this set. However, before breaking the atomic operations into elementary units (loads/stores), we first try and maintain more coarse, "higher-level", operations. This is done by providing the system with user-defined atomicity constraints that are guaranteed to be maintained during exploration. For the collector, we make building blocks $C1$ and $C2$ of Table 1 a single atomic operation, expressed as $readMarkTarget = [C1\,C2]$. That is, the reading of the field and the marking of the object who is pointed to by this field are executed atomically. For the mutator, we make the increment of the mutator count and the logging of the object execute atomically, expressed as $incAndLog = [M3\,M5]$.

Given these constraints over the building blocks, we ran the system and the least atomic algorithm found is shown in Fig. 4. In this paper, in all figures presenting algorithms, we use *atomic* to denote an initial atomicity constraint provided by user, and *detectedAtomic* to denote atomicity constraints detected as required by the system. The details of this execution can be seen as run (a) of Table 2.

The *detectedAtomic* constraints in the resulting algorithm effectively suggests that the mutator should store the target object and read the wavefront atomically. Similarly, the collector should read the field and set its progress atomically.

Initially, we were surprised that the setting of the field and the reading of the wavefront in the mutator had to be atomic, even more so because the system had found the processing of the field was done atomically in the collector, so the mutator could not preempt the collector during the processing of a field. To figure out why *detectedAtomic* in the mutator was necessary, we examined a counterexample generated by SPIN for an algorithm without this *detectedAtomic*. It indicated a subtle interleaving where an incorrect decrement of a mutator count was taking place. The system was helpful in quickly discarding such candidates and finding the correct alternatives.

Our automatic exploration procedure is exhaustive. Thus, it is guaranteed that the algorithm of Fig. 4 is the least atomic algorithm that the system could find under the provided set of building blocks and user-provided atomicity constraints. However, the user may still be unsatisfied (as she should be) with the current result. She may want to derive an algorithm with fewer atomicity constraints.

5.2 Manual Step: Adding New Building Blocks

At this point, new insights and new building blocks are needed in order to continue the derivation and a human is required to generate an insight. But how can we generate an insight which is as generic as possible, that is, not garbage collection dependent? We asked a natural question of what would happen if more state is injected into the system? Can it be used to reduce atomicity? And if so, how should the state be introduced?

To answer these questions, we distinguish between two types of building blocks:

- Core blocks - describe the core operations performed by the collector and the mutator, such as the collector reading and marking an object ($readMarkTarget$) or the mutator storing a pointer in the heap ($M2$). One can think of these as operations which cannot be avoided and exist even in sequential collectors where the mutator cannot preempt the collector during marking.

Collector		
BB#	**Building Block**	**Meaning**
$C4_S$	$o.S_{fld} \leftarrow true$	notify field scan starts
$C4_E$	$o.E_{fld} \leftarrow true$	notify field scan ends
Mutator		
BB#	**Building Block**	**Meaning**
$M6_S$	$val \leftarrow o.S_{fld}$	check if field scan started
$M6_E$	$val \leftarrow o.E_{fld}$	check if field scan ended

Table 3. Refined building blocks, replacing the blocks $C4$ and $M6$ of Table 1, and recording both the beginning and the ending of the wavefront progress over a field.

- Auxiliary blocks - describe the operations on collector metadata such as the mutator count field in each object, or the progress of the collector through the heap via modifying the wavefront.

The next step can be thought of as adding new metadata, that is, new auxiliary building blocks. The question of where to add this metadata is a central one. Our insight was to add state which captures the history of the execution of the core operations, that is, the starting and the ending of each core operation.

One can think of the current per-field wavefront bit as a way of notifying the mutator that the field has been read by the collector. However, the role of this bit relies on the atomicity of the collector sequence. When the sequence is atomic, the wavefront variable is observed by the mutator if and only if the field has been scanned by the collector. If the collector sequence becomes non-atomic, the wavefront variable can no longer convey the same semantics as it is updated separately from the scanning of the field.

Therefore, we add state to capture the history of the core operations. For each field in an object we introduce a new bit per field S_{fld}, which notifies the mutator when the collector starts processing the field fld. The original wavefront bit retains its semantics that when it is set, the field has been read by the collector. To reflect the fact that we now have two fields, we rename the original wavefront bit from $\mathcal{W}_{fld}$ to E_{fld}. These two fields make the progress of the collector more precisely observable to the mutator. The new state also implies that we need new auxiliary building blocks for reading and writing this state (the collector writes and the mutator reads this state). The additional blocks are shown in Table 3.

In the future, we would like to see the system automatically apply various transformation rules such as the one described here (adding state in a specific place). However, at this stage, the addition of auxiliary state is done manually. Next, we proceed by showing how to use these building blocks in order to further reduce atomicity.

5.3 Exploration: Another Try

Before proceeding, we add an ordering constraint over the collector's blocks which reflects our insight about notifying the start and the end of a core operation:

$$CO1:\ o.S_{fld} \leftarrow true < ReadMarkTarget < o.E_{fld} \leftarrow true$$

In addition, we also added the data-dependent constraints that decrementing mutator count is dependent on the value of E_{fld} (i.e. mutator building block $M4$ is enabled only if E_{fld} is $true$).

We ran the system again using the additional building blocks and the $CO1$ constraint. The details of this invocation can be seen in Table 2 as run (b). Exploration of the algorithm space yielded two least atomic correct algorithms (i.e. where no *detectedAtomic* was required). Fig. 5 shows one of these algorithms. The other algorithm is similar and is obtained by swapping the first two lines of `mutate`.

```
mutate_5()
old ← obj.fld
e ← obj.E_fld
obj.fld ← new
s ← obj.S_fld
atomic
  [ if(s) new.MC++
  [ if(s) cand ← cand ∪ {new}
if(e) old.MC−−
```

```
markStep_5()
o.S_fld ← true
atomic
  [ dst ← o.fld
  | if (dst ≠ null ∧ ¬dst.mark)
  |   dst.mark ← true
  [ else dst ← null
o.E_fld ← true
return dst
```

Figure 5. The least atomic algorithm that can be derived from the building blocks of Table 1 refined by the blocks of Table 3.

The following three mutator constraints describe the `mutate` procedure of the two least atomic algorithms just discovered:

$$\begin{array}{ll} DMC1: & end \leftarrow o.E_{fld} < o.fld \leftarrow new \\ DMC2: & o.fld \leftarrow new < start \leftarrow o.S_{fld} \\ DMC3: & incAndLog < if(e)\ old.MC{-}{-} \end{array}$$

The first two constraints state that the mutator processes an object field in the opposite order from the collector. The third constraint is between mutator operations. It is an interesting one because it says that although `addOrigins` is atomic, the system requires the logging of an object to occur before a potential mutator count decrement of another or same object. This is due to the fact that although `addOrigins` cannot be preempted by `mutate`, it can still preempt `mutate`. In fact, we also performed another experiment where `addOrigins` could not preempt `mutate` and `mutate` could not preempt `addOrigins` by using a global synchronization barrier in the skeleton before each `addOrigins` starts. With this more global constraint we actually derived (not surprisingly) more correct algorithms. That is, we were able to discover the first two constraints $DMC1$ and $DMC2$, but not $DMC3$. The experiment suggested that what look like small changes in the skeleton can affect algorithm correctness in subtle ways. Reasoning about these changes manually in the presence of concurrency is very difficult for a human. Moreover, finding all possible correct solutions after a small change is made and expressing all of the resulting solutions concisely as a set of constraints is nearly impossible to correctly reason about manually.

5.4 Abstracting State

At this stage, our exploration has produced a non-atomic `mutate` and `markStep` in the building blocks we have provided, but with an atomic `addOrigins`. Before we continue searching for less atomic algorithms, a question to ask is: can we achieve the same non-atomicity with less auxiliary state and if so, is there some property of the algorithms that is affected by using less state?

We decide to try and answer this question by experimenting with our system. Naturally, we concentrate on auxiliary state and abstracting the collector progress. Abstracting collector progress simply means the mutator does not observe such progress precisely. We explore three corner cases of this abstraction, although more variations could be introduced. The results of these three experiments are shown in Table 2:

1. Removal of S_{fld} state from every field is shown as run (d).
2. Removal of E_{fld} state from every field is shown as run (e).
3. Removal of S_{fld} and E_{fld} from every field is shown as run (c).

Removal of state can be thought of as the mutator building blocks $M6_S$ and $M4_E$ simply returning *true* or *false*. That is, they are unable to observe the collector progress precisely and are forced to return a safe approximation, which for $M6_S$ is *true* and for $M4_E$ is *false*.

It may seem that this process of abstracting state contradicts the previous step of introducing additional state by suggesting to now remove the same state from every field. However, there is a fundamental difference. Earlier, both the increment of the mutator count and the decrement depended on the *same* bit $\mathcal{W}_{fld}$. By splitting this bit into two bits S_{fld} and E_{fld}, the increment and decrement no longer depend on the same state. Therefore, even when we remove the S_{fld} bit from every field, the result is not the same as having only the $\mathcal{W}_{fld}$, because the resulting mutator incrementing operations fundamentally depend on different states.

The removal of only the S_{fld} bit, as suggested by the first case, led to the generation of 14 correct and least atomic variations. That is, the system did not detect that more atomics were necessary than those already provided in the building blocks. All 14 variations can be expectedly described with the two constraints presented earlier: $DMC1$ and $DMC3$. The constraint $DMC2$ is not necessary anymore because $M6_S$ always returns true in this case.

When we removed the E_{fld} bit only, as suggested in the second case, the system generated only one correct least atomic variation. This result is represented by the constraint $DMC2$ presented earlier, which is expected because $M4_E$ always returns false and hence the constraints $DMC1$ and $DMC3$ are unnecessary.

Note that unlike in Fig. 4 where we needed *detectedAtomic*, in the two examples so far, we found algorithms which do not require such a constraint. This is interesting because they use the same state size, namely a bit per field. As mentioned, the key difference is the meaning of this state and what actions are triggered by the mutator upon reading it.

In our third experiment, in which both bits were removed, the system found two correct least atomic variations. The system did not derive any required constraints. That is, the algorithms are strictly described by their input data-flow constraints. Effectively, the algorithms always increment the mutator count, and never decrement it.

Certainly, even without running the system, we could have deduced the derived constraints from these three runs by simply eliminating the corresponding constraint when the building block is known to return a constant value (e.g. $true$ or $false$). However, the particular searching order depends on the expertise of the user and hence it is possible the user simply did not reach the original three constraints ($DMC1$, $DMC2$ and $DMC3$) in their design, but managed to think of a variation which had the S_{fld} bit only.

It is interesting to mention that the third variation with the least state (i.e. both bits removed) is a symbolic algorithm which is very similar to the original Dijkstra algorithm. However, it is well-known that the order of statements (that is, the store on the new target and the coloring of the target) in the original Dijkstra algorithm is crucial to correctness. The reason why the order is critical there is due to the fact that while in the write barrier (e.g. `mutate`), it is possible for one collection cycle to finish and another collection cycle to start. Such an interleaving is not possible in our system and in most practical systems which require a stopping phase for each thread in order to mark their root set.

5.5 Final Step: Reducing Atomicity In Candidate Processing

We resume our search for a non-atomic collector. Our starting point are the building blocks described in Table 3 and the results obtained in Section 5.3. Recall that the mutator logging of objects and collector's `addOrigins` are still atomic operations (building blocks $M5$ and $C3$). Our next task is splitting these blocks into separate smaller building blocks. In particular, we first manually make `addOrigins` non-atomic by removing the atomic section around it and placing it inside `addOriginStep`, so now the whole of `addOriginStep` is atomic, but `addOrigins` is not. The splitting amounts to simply removing an atomicity constraint.

Collector		
BB#	**Building Block**	**Meaning**
$C3_A$	$o \leftarrow cand[i]$	read object from $cand$ set
$C3_B$	$o.inLog \leftarrow false$	reset $inLog$ bit
$C3_C$	$if\ (\neg o.mark \wedge o.MC > 0)$ $o.mark \leftarrow true$	mark object
Mutator		
BB#	**Building Block**	**Meaning**
$M5_A$	$if\ (v1)\ cand[nextcand{+}{+}] \leftarrow o$	add object to $cand$
$M5_B$	$if\ (v1)\ o.inLog \leftarrow true$	set $inLog$ bit
$M5_C$	$if\ (v1)\ v2 \leftarrow o.inLog$	read $inLog$ bit

Table 4. Refined building blocks, replacing the blocks $C3$ and $M5$ of Table 1, explicitly handling the candidate set $cand$.

Collector		
BB#	**Building Block**	**Meaning**
$C5$	$gc\_state \leftarrow phase$	write gc_state. $phase$ is either $trace$ or $expose$
Mutator		
BB#	**Building Block**	**Meaning**
$M7$	$if\ (v1)\ v2 \leftarrow gc\_state$	read gc_state
$M8$	$if\ (v1)\ o.MC \leftarrow (val = trace)\ ?\ o.MC + 1 : max$	increment or overflow the mutator count.

Table 5. Additional building blocks to Table 4.

Therefore, the new collector building blocks are basically those of `addOriginStep`. These refined building blocks are shown in Table 4. Similarly, the building block $M5$ is split into several smaller pieces manually. In fact, these pieces of $M5$ were always present, but were not shown earlier as logging was executed atomically. In addition, we also remove the user provided atomicity constraint of *incAndLog*, so now $M3$ (mutator increments) and mutator logging are no longer constrained to execute atomically.

Therefore, the only atomicity constraints left in the mutator are those forming the individual building blocks. For example, in block $M5_A$, we log and then increment $nextcand$ atomically. In the collector, as before, $readMarkTarget$ forms a single coarse-level building block.

We ran the system with these new building blocks and fewer constraints. This run is depicted as run (j) in Table 2. Unfortunately the exploration could not find a correct algorithm. Despite the fact that exploration was not successful, there is still a key point to be made. The main reason we were able to explore over $350,000$ variations in less than 40 minutes is because we used two of the ordering constraints that we discovered in Section 5.3: $DMC1$ and $DMC2$. In this section our final goal is to have no *detectedAtomic* between building blocks and therefore we knew that constraints $DMC1$ and $DMC2$ must hold if the result is to be non-atomic (as we already know they are required for dealing with the interference between the `mutate` and `markStep`). In this sense, we are using the system in a feedback-directed manner, inserting a constraint from a previous stage in order to direct the search more accurately. In our experience, such informed guidance is vital in obtaining realistic running times.

At this point, the system has deduced that even with an atomic `addOriginStep`, `mutate` and `markStep`, it is still not possible to produce even a single correct algorithm.

Again, an insight is clearly needed. We manually studied why an algorithm fails and figured out the problem to be due to mutator counts going up and down while interleaving with the execution of `addOrigins`. It is possible to construct an example where `addOrigins` never terminates, because an object is continuously being re-logged during its execution. To solve this problem, we look for a way to guarantee progress of `addOrigins` and avoid a potentially infinite log while still guaranteeing safety.

We decide to follow the same pattern we took in Section 5.2 and add some auxiliary control state to the collector. Therefore, all we add is a flag (a variable) which signals whether the collector is in the marking phase or whether it is in the `addOrigins` phase. We refer to this variable as gc_state and the corresponding building blocks to read and write this variable are shown in Table 5. For the mutator, building block $M3$ which performs increments is subsumed by block $M8$. In $M8$, the mutator checks whether the gc_state has been set to $expose$ and if so, the mutator count overflows and hence can no longer be decremented. To fundamentally understand how gc_state came into existence, we note that ideally, we would have added $start$ and end bits to each log field and explored the possible results. The gc_state variable is in fact an abstraction of the collector progress information through the log (similar to the abstraction of the collector progress through the heap discussed in Section 5.4).

We ran the system again with the additional building blocks. Run (i) in Table 2 shows the details of this run. This run was split into four parts and we report the maximum time it took for a single part to complete. Note that the time it took for this run to complete is much longer than the other runs because the model exploration requires memory which is close to the physical memory of the machine. It took 7.8GB for the checking of the largest model and the available physical memory was 8GB. The exploration found six correct algorithms. For two of these algorithms, the system did not require any usage of *detectedAtomic* at all, both for the mutator part and for the collector part. One of these two non-atomic algorithms (instantiated inside the skeleton) is shown in Fig. 6. The second algorithm can be obtained by swapping the first two building blocks of `mutate`. Note that `addOrigins` has no enclosing atomic section as well. The state variable gc_state is set to tracing whenever the collector begins the tracing phase, and is set to expose before `addOrigins` begins. It is set back to tracing when `addOrigins` finishes.

Our system dictated that all six correct algorithms must comply with the following derived mutator ordering constraints:

$$\begin{array}{ll} DMC4: & M8 < M5_C \\ DMC5: & M5_A < M4 \end{array}$$

In fact these two constraints are simply refinements of constraint $DMC3$ presented earlier. The first constraint indicates that an increment/overflow of a mutator count should occur before checking whether the object is in the candidate set. The second one states that the store of the object in the candidate set should occur before the decrement of its count.

The other four algorithms look like the one in Fig. 6, except that $M5_B$ is moved either after $M5_A$ or after $M4$. The system states that if such movement of $M5_B$ occurs, then an atomic section is required around $C3_B$ and $C3_C$ (i.e. *detectedAtomic*).

As in Section 5.4, we also experimented with the system by abstracting the progress of the collector. That is, by removing either S_{fld}, E_{fld} bits or both from every field. Run (h) in Table 2 reflects the removal of E_{fld} and run (f) reflects the removal of both. Run (h) indicated that no *detectedAtomic* was necessary in both the mutator or the collector, so the only constraints required are the initial data-dependency constraints. For run (f), the system detected one derived constraint $M8 < M5_A$, indicating that the increment of the mutator count had to occur before the logging of the object. Case (g) (removal of S_{fld} only) timed out.

5.6 Future Steps

Although the algorithm obtained in Fig. 6 did not require *detectedAtomic* constraints, it still has $atomic$ inside the code (due

```
markStep(obj,fld) {
  o.S_fld ← true
  atomic
  [ dst ← o.fld
  [ if (dst ≠ null ∧ ¬dst.mark)
  [ dst.mark ← true
  [ else  dst ← null
  o.E_fld ← true
  return dst
}
```

(MRK) Collector tracing

```
addOrigins() {
  gc_state ← expose
  for (i = 0; i < |nextcand|; i++) {
    dst ← addOriginStep(i)
    if  (dst ≠ null)
      pending ← ∪ {dst}
  }
  gc_state ← trace
}

  addOriginStep(i) {
    dst ← cand[i]
    dst.inLog ← false
    atomic
    [ if (dst.MC > 0 ∧ ¬dst.mark)
    [   dst.mark ← true
    [ else  dst ← null
    return dst
  }
```

(ORG) Collector adding origins

```
mutate(source, fld, new) {
old ← source.fld
e ← source.E_fld
source.fld ← new
s ← source.S_fld
if (s)  {
  val ← gc_state
  atomic
    [ if (val = trace)
    [     new.MC++
    [ else
    [     new.MC = max
  val ← new.inLog
}
if (¬val)  {
  new.inLog ← true
  atomic
    [ cand[nextcand++] ← new
}
atomic
  [ if(e)  old.MC--
}
```

(MUT) Mutator barriers

Figure 6. The most fine-grained algorithm derived in our framework.

to the designer requesting them). However, note that there is no building block which is executed atomically that is accessing two or more variables where both variables can be read *and* written by both the collector and the mutator. The next step in the exploration would be to remove these final atomicity constraints and check the algorithm. However, the algorithm in Fig. 6 already requires 7.8GB and our machine had 8GB, so it would be impossible to check the resulting less atomic algorithm. It would be interesting to continue the exploration either after applying various engineering optimizations to the model to make it smaller or simply after more memory is obtained.

Moreover, although our proofs and derivations are for algorithms allocating black, this restriction is only necessary because we are more general than practical algorithms which perform a stop-the-world phase for stack rescanning. Assuming such a synchronization phase undoubtedly will lead to more interesting variations as well as being able to handle white objects. Further, we would be interested in seeing this work extended for multiple mutators or write barriers other than counting as presented in [24]. This would certainly require more sophisticated abstraction as well more memory to verify the results.

6. Verifying Derived Algorithms

We verify the derived algorithms using model checking with abstraction. Automatically verifying arbitrary concurrent garbage collection algorithms, with all of their details, is a challenging task. The alert reader may therefore find the above claim surprising. However, the reason that our verification attempt is successful, is that we operate within the boundaries of our limited framework.

In particular, we are making the following assumptions:

- The algorithm handles a single collector and a single mutator.
- The implemented algorithm is an attempted implementation of a counting algorithm, where the counting threshold is known.
- The algorithm skeleton is fixed, and the operations performed by the skeleton are known to be correct. For example, we assume that basic stop-the-world tracing is implemented correctly (i.e., the trace procedure marks all the objects that are reachable from the pending set when it executes without interruptions).
- The algorithm uses a synchronization barrier before moving to the sweep phase, so mutations are required to terminate before the collector ends the mark phase.

Therefore our procedure verifies that using the derived write-barrier and collector-step inside a given (correct) skeleton yields a correct algorithm. In this section, we describe key aspects of the verification process.

6.1 Verification Problem: Safety of the Derived Algorithms

The correctness of the derived algorithms is specified by the following safety invariant:

DEFINITION 6.1 (Safety Invariant). *When the last execution of* `addOrigins` *terminates, that is, before* `sweep` *starts, all reachable objects are marked.*

Using the safety invariant as the verification goal requires reasoning about reachability properties in arbitrary heaps undergoing arbitrary mutations while taking into account the interference between the mutator's write barrier and the collector's tracing of the heap and processing of the log. Reasoning about reachability properties in such a setting is very challenging. However, we avoid the need to about reason reachability properties using the following observation: For the safety invariant to be violated, there must exist an object o that is reachable but not marked. If such an object o exists, there exists an object o' such that o' is directly pointed to by a black object, but o' is not marked.

This observation allows to establish that the safety invariant holds by verifying that the following local safety invariant holds

DEFINITION 6.2 (Local Safety Invariant). *When the last execution of* `addOrigins` *terminates, that is, before* `sweep` *starts, every object which is pointed to by a black object, is marked.*

We verify that the local safety property holds by non-deterministically selecting a *tracked object* and checking whether any interaction of mutator and collector can cause this object to violate the local safety invariant. This selection is similar to the choice of a single tracked object in e.g., [7, 25, 13].

6.2 Abstraction for Model Checking

We verify that the local safety invariant holds using model checking with abstraction. Our abstraction represents an unbounded number of heap locations by a bounded abstract representation. We partition the heap into equivalence classes based on properties of heap locations. Intuitively, our abstraction partitions the heap in a way that distinguishes several classes of locations, including:

- locations that have already been read by the collector (scanned locations).
- locations marked by the collector (marked locations)
- header locations with different mutator counts.

Our abstraction is sound, so when we successfully verify an algorithm, it is indeed guaranteed to be correct. However, when we are unable to establish the correctness of an algorithm under our abstraction, it is possible that the algorithm is still correct, but our abstraction is insufficient for showing its correctness (hence we may have missed some correct algorithms).

To maintain sufficiently precise abstraction we identify fields that were read by the collector (scanned) and then updated by the mutator to point to the tracked object. We refine the abstraction of the scanned location by keeping these fields distinct, while the mutator count (MC) of the tracked object has not overflowed. Tracking these fields allows us to precisely handle decrements of the mutator count. A key observation that we are using in order to bound the number of these locations, is that there are only k relevant pointers that can be installed pointing to the tracked object before its count overflows. Because we assume that k is known, we are able to precisely track only a bounded number of specific locations, while aggressively abstracting the rest of the heap.

Technically, we use the framework of [20] to describe the heap and properties of heap locations using first-order logic. We then hand-code the abstraction into SPIN to achieve a more efficient implementation.

7. Conclusion

We present a framework for synthesizing *provably correct* counting based concurrent mark and sweep collection algorithms. Our framework utilizes a user-provided set of building blocks to automatically explore a space of algorithms, using model checking with abstraction to verify algorithms in the space. Using our framework, we were able to discover several interesting fine-grained algorithms. In future work, we plan to investigate the possibility of applying our techniques in other domains. We believe that the key reason for the success of our framework is that it found a good balance between the tasks performed by a human and the tasks performed by the machine.

References

[1] BAR-DAVID, Y., AND TAUBENFELD, G. Automatic discovery of mutual exclusion algorithms. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing* (2003).

[2] BARABASH, K., OSSIA, Y., AND PETRANK, E. Mostly concurrent garbage collection revisited. In *Proceedings of the 18th ACM conference on Object-oriented programing, systems, languages, and applications* (2003).

[3] BEN-ARI, M. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst. 6*, 3 (1984).

[4] BIRKEDAL, L., TORP-SMITH, N., AND REYNOLDS, J. C. Local reasoning about a copying garbage collector. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages* (2004).

[5] BOWMAN, H., DERRICK, J., AND JONES, R. E. Modelling garbage collection algorithms. In *Proceedings of International Computing Symposium* (1994).

[6] BURDY, L. B vs. Coq to prove a garbage collector. In *the 14th International Conference on Theorem Proving in Higher Order Logics: Supplemental Proceedings* (2001).

[7] DAS, M., LERNER, S., AND SEIGLE, M. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (2002).

[8] DEMMERS, A., WEISER, M., HAYES, B., BOEHM, H., BOBROW, D., AND SHENKER, S. Combining generational and conservative garbage collection: framework and implementations. In *Proceedings of the 17th ACM symposium on Principles of programming languages* (1990).

[9] DEWAR, R. B. K., SHIRAR, M., AND WEIXELBAUM, E. Transformational derivation of a garbage collection algorithm. *ACM Trans. Program. Lang. Syst. 4*, 4 (1982).

[10] DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM 21*, 11 (1978).

[11] GRIES, D. An exercise in proving parallel programs correct. *Commun. ACM 20*, 12 (1977).

[12] GRIES, D. Corrigendum. *Commun. ACM 21*, 12 (December 1978), 1048.

[13] HACKETT, B., AND RUGINA, R. Region-based shape analysis with tracked locations. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages* (2005), ACM.

[14] HAVELUND, K. Mechanical verification of a garbage collector. In *Fourth International Workshop on Formal Methods for Parallel Programming: Theory and Applications* (1999).

[15] JACKSON, P. B. Verifying a garbage collection algorithm. In *Theorem Proving in Higher Order Logics, 11th International Conference* (1998).

[16] MASSALIN, H. Superoptimizer: a look at the smallest program. In *the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* (1987).

[17] PAULSON, L. *Isabelle: A Generic Theorem Prover*, vol. 828 of *Lecture Notes in Computer Science*. 1994.

[18] PRENSA NIETO, L., AND ESPARZA, J. Verifying single and multi-mutator garbage collectors with Owicki/Gries in Isabelle/HOL. In *Mathematical Foundations of Computer Science* (2000).

[19] RUSSINOFF, D. M. A mechanically verified incremental garbage collector. *Formal Aspects of Computing 6*, 4 (1994).

[20] SAGIV, M., REPS, T., AND WILHELM, R. Parametric shape analysis via 3-valued logic. *ACM Trans. on Prog. Lang. and Systems 24*, 3 (2002).

[21] SOLAR-LEZAMA, A., RABBAH, R. M., BODÍK, R., AND EBCIOGLU, K. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (2005).

[22] TAKAHASHI, K. *Abstraction and Search in Verification by State Exploration*. PhD thesis, University of Tokyo, Jan. 2002.

[23] VECHEV, M. *Derivation And Evaluation Of Concurrent Collectors*. PhD thesis, University of Cambridge, 2007.

[24] VECHEV, M. T., YAHAV, E., AND BACON, D. F. Correctness-preserving derivation of concurrent garbage collection algorithms. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (2006).

[25] YAHAV, E., AND RAMALINGAM, G. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM conference on Programming language design and implementation* (2004).