Using Garbage Collection in Model Checking

Radu Iosif[1] and Riccardo Sisto[1]

Dipartimento di Automatica e Informatica, Politecnico di Torino
corso Duca degli Abruzzi 24, 10129 Torino, Italy
iosif@athena.polito.it, sisto@polito.it
http://www.dai-arc.polito.it

Abstract. Garbage collection techniques have become common-place in actual programming environments, helping programmers to avoid memory fragmentation and invalid referencing problems. In order to efficiently model check programs that use garbage collection, similar functionalities have to be embedded in model checkers. This paper focuses on the implementation of two classic garbage collection algorithms in dSPIN, an extension of the model checker SPIN which supports dynamic memory management. Experiments carried out show that, besides making a large class of programs tractable, garbage collection can also be a mean to reduce the number of states generated by our model checking tool.

1 Introduction

Applying finite-state verification techniques, such as model checking, to concurrent and distributed software systems looks attractive because they are capable of detecting very subtle defects in the logic design of these systems. Nevertheless, the transition of these techniques from research to actual practice is still slow. One of the reasons is that current finite-state verification tools still adhere to a static representation of systems, while programming environments become more and more dynamic. It is necessary to distinguish here between static and dynamic program information, the former referring to information that can be known at compile-time using static analysis techniques (e.g., data flow analysis), while the later refers to information that occurs only at run-time. In fact, many of the optimizations performed in compilers (and lately in software verification tools [3]) are based on the attempt to over-approximate run-time information using static analysis techniques. Even if static analysis proves to be useful in reducing the size of finite-state program models, there are still cases in which effective reductions can be achieved only at the expense of very complex and time-consuming analysis. Such cases involve pointer analysis [2] which produces a conservative approximation of a pointer alias set that is, the set of objects it may point to at run-time. Besides being very complex, the analysis results can still be too large to be used in finite-state verification.

An alternative approach to software verification involves the representation of dynamic program information directly into the model checking engine. In order to do that, we have extended the model checker SPIN [7] with a number of dynamic features, among which:

– memory references (pointers),
– dynamic object creation and deletion,
– function declaration and call,
– function code references (function pointers).

The extension is called dSPIN (dynamic SPIN) and has been reported in [5]. Even if dSPIN remains a general-purpose model checker, it is intended especially for software verification, allowing for an easy translation of high-level object-oriented programming languages (Java, C++) into its input language, which is a dialect of PROMELA. The size of the transition system is reduced first using a light-weight pointer analysis, and then applying common finite-state reduction techniques such as: partial order reductions, state compression, symbolic representation, etc.

A further step towards improving model checking of software systems is the introduction of garbage collection techniques in dSPIN. This work is motivated by the widespread use of garbage collection in real-life software systems, especially the ones written in Java, where this is the default memory management policy. Indeed, when unused memory is not reclaimed, the program state space may experience an unbounded growth that makes analysis impossible. For example, let us assume that `C` represents the name of a class declaration in the following fragment of Java code:

```
C x;
while (true) {
    ...
  x = new C( );
    ...
}
```

If no garbage collection is performed during model checking, every iteration of the (possibly infinite) loop will add new states to the program state space. When the state space is explored in the depth-first order, this leads to a very fast growth of the state space that exhausts the system resources before any useful results can be given. Instead, when the program model is checked using garbage collection, the first state of the second iteration will (most probably) match the first state of the first one and the loop need not be re-explored.

Without using garbage collection, a solution can be explicit memory deletion. However, using explicit deletion has two drawbacks. On one hand, if the source program does not use explicit deletion statements, inserting such statements into the program model requires expensive pointer analysis. On the other hand, explicit deletes may result in useless changes of the memory configuration, which greatly increases the program state space. Embedding garbage collection into the model checker proves to be quite easy and it has the advantage of eliminating both, the need for complex pointer analysis and dangerous memory fragmentation at the expense of only a small run-time overhead.

It has recently come to our attention that at least one other group is working on implementing garbage collection in a Java byte-code model checker, the Java PathFinder tool [6].

The paper is organized as follows: Section 2 recalls background concepts on dSPIN and garbage collection, Section 3 describes the implementation of two collection algorithms in dSPIN, Section 4 reports a number of experiments that have been carried out and Section 5 concludes.

2 Background

In this section we recall background concepts used throughout the paper. In particular, we present two classic collection algorithms and discuss the data layout in dSPIN. A detailed report on dSPIN can be found in [5].

In what follows, we denote by *garbage* [1] any heap-allocated object that is not reachable by any chain of pointers from program variables. The memory occupied by garbage should be reclaimed for use in allocating new objects. This process is called *garbage collection* and is performed by the program run-time system, in our case the model checking engine.

2.1 Reference Counting Collection

Most of the garbage collection algorithms identify the garbage by first finding out what is reachable. Instead, it can be done directly by keeping track of how many pointers point directly to each object. This is the *reference count* of the object and needs to be recorded with each object. More precisely, we need to keep the reference count for all objects in a separate table, called the *reference table*. Figure 1 shows the general structure of reference counting collection algorithms [1]. Whenever an object reference `p` is stored into a pointer variable `v`, the object

```
procedure GCdec(p)
begin
  reference_table [p] --;
  if (reference_table [p] == 0)
  then begin
    for each field f of object p
      GCdec(f);
    delete(p);
  end
end GCdec
procedure GCinc(p)
begin
  reference_table [p] ++;
end GCinc
```

Fig. 1. Reference Counting Collection

reference count is incremented by a call to `GCinc(p)` and the reference count of what `v` previously pointed to is decremented by a call to `GCdec(v)`. If the

decremented count of an object reaches zero then the object is deleted and the reference counts of all other objects pointed to by its fields are decremented by recursive calls to `GCdec`.

Reference counting seems simple and attractive but there is a major drawback: cycles of garbage cannot be reclaimed. Let us consider for example a cyclic list of objects that is not anymore reachable from program variables. Every cons of the list will still have the reference count at least one, which prevents it from being collected.

Considering that a pass of the reference counting collector has collected $N_{collected}$ objects out of a total of $N_{reachable}$ reachable, the time spent can be evaluated as $t_{delete} \times N_{collected}$, where t_{delete} stands for the average object deletion time. The worst-case time of a reference counting collection occurs when all reachable objects are collected i.e., $t_{delete} \times N_{reachable}$. This corresponds to the case in which all reachable objects have the reference count equal to one, which makes the collector reclaim them all.

Despite the previously mentioned problem, regarding the impossibility of reclaiming cyclic-structured data, the cost of a reference counting collection is quite small which makes the algorithm suitable for use in model checking.

2.2 Mark and Sweep Collection

Program pointers and heap-allocated objects form a directed graph. Program variables are the roots of this graph, denoted in what follows by the set `ROOTS`. A node `n` in the graph is reachable if there is a path of directed edges leading to `n` starting at some root `r` $\in$ `ROOTS`. A graph-search algorithm, such as depth-first search, marks all nodes reachable from all roots. Any node that is not marked is garbage and must be reclaimed. This is done by a sweep of the heap area, looking for nodes that are not marked. As said, these nodes are garbage and they are deleted. The sweep phase should also clear the marking of all nodes, in preparation for the next collection. Figure 2 shows the mark and sweep collection algorithm [1]. The marking of all objects are kept in a global *mark table* used by both the `GCmark` and `GCsweep` procedures.

In order to evaluate the cost of a mark and sweep collection, let us consider that there are $N_{reachable}$ reachable objects. We denote by N_{max} the upper bound of the heap that is, the maximum number of objects that can be allocated. The time taken by the mark phase is proportional to the number of nodes it marks that is, the amount of reachable data. The time taken by the sweep phase is proportional to the size of the heap. Consequently, the overall time of the garbage collection is $t_{mark} \times N_{reachable} + t_{delete} \times (N_{max} - N_{reachable})$, where t_{mark} is the object marking time and t_{delete} is the object deletion time, introduced in the previous section. Once again, this is a worst-case estimation because in practice, only the number of unreachable existent objects needs to be deleted by the collector. As this number depends tightly on the program structure, we have chosen to over-approximate it with the maximum number of unreachable objects $N_{max} - N_{reachable}$.

```
procedure GCmark(r)
begin
  if (! mark_table [r])
  then begin
    mark_table [r] = true;
    for each field f of object r
      GCmark(f);
  end
end GCmark
procedure GCsweep()
begin
  for each object p
  if (mark_table [p])
  then begin
    mark_table [p] = false;
    delete(p);
  end
end GCsweep
```

Fig. 2. Mark and Sweep Collection

2.3 Overview of dSPIN

In order to make this paper self contained, we need to recall some of the extensions that have been implemented in dSPIN, in particular the ones concerning dynamic memory management. In dSPIN, memory can be dynamically allocated and deleted by means of explicit `new` and `delete` statements. Dynamically allocated areas will be denoted in what follows as *objects*. The mechanism to handle objects in dSPIN is called *pointer*, as in most programming languages. Semantically, dSPIN pointers resemble to Java references that is, they can be assigned and the objects they point to can be accessed (dereferenced), but they cannot be used in arithmetic operations, as it is the case in C. A complete description of the syntax and semantics of these language constructs can be found in [8].

In dSPIN, the objects that are dynamically allocated reside on a contiguous memory zone called the *heap area*. Newly created objects are added at the end of the heap area. After deleting an existing object, the heap area is compacted in order to avoid memory losses caused by fragmentation. The object retrieval information is kept into two tables called the *offset table* and the *size table*. The first table holds the offset with respect to the beginning of the heap area, while the second one holds the actual size of the object. In addition, a garbage collector must be able to operate on objects of all types. In particular, it must be able to determine the number of fields in each object, and whether each field is a pointer. Type information is kept for each object in the *type table*. A pointer to an object encodes an integer value which is used to index the three tables. Figure 3 depicts the run-time representation of objects in dSPIN.

Explicit type information is needed at run-time because the input language of dSPIN allows for free pointer conversion. In other words, conversion between

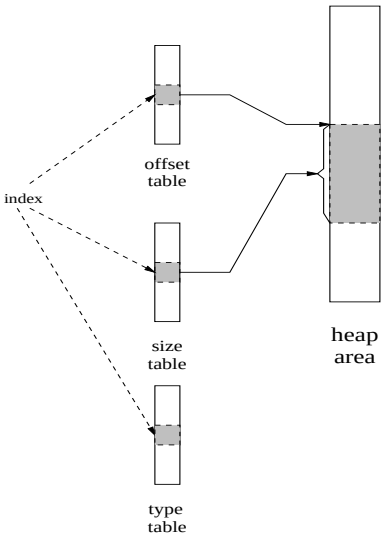

Fig. 3. The Data Layout in dSPIN

any two pointer types is allowed, which makes static type evaluation almost impossible. Let us point out that type information need not be stored into the state vector when performing model checking. The offset, size and type tables are global data structures that are modified only by the allocation and deletion statements. As will be discussed in the following, this information needs to be recorded on the state stack in order to allow the model checker to unwind allocation/deletion forward moves. The memory overhead introduced by garbage collection depends only on the maximum stack depth.

3 Implementation Issues

In this section we present issues related to the implementation of the previously presented collection algorithms, in the dSPIN model checker. In particular, we address aspects related to the compatibility between these algorithms and the model checking run-time environment. As the latter differs from common run-time systems, the way its particularities may interact with garbage collection are considered.

A first point that needs attention is the possibility of duplicating regions of the model checker state space when using garbage collection. For example, let us consider the situation in which the verifier generates a state A in which some objects have become garbage. Let us assume that, like in usual run-time environments, the garbage collector does not necessarily run immediately after this state has been generated. In consequence, all its direct successor states, let us call them $B_1, B_2, \ldots, B_n$ will maintain the garbage. If, at a later point during verification, the model checker generates the state A' that differs from A only

by the absence of garbage, the states A and A' will not match. Consequently, none of the direct successors of A', let us call them $B'_1, B'_2, \ldots, B'_n$ will match the states $B_1, B_2, \ldots, B_n$ respectively. This results in an unwanted duplication of states, because of garbage collection.

The solution is to run the collection as soon as possible in order to avoid states containing any garbage at all. In order to do that, we distinguish three kinds of statements that may produce garbage in dSPIN:

1. pointer *assignment*; the object previously pointed to by the pointer may become garbage.
2. *exit* from function or local scope; the objects pointed to by local pointers may become garbage.
3. explicit *delete*; the objects pointed to by the deleted object fields may become garbage.

Garbage collection is invoked atomically, as part of the transitions fired by the statements above. The model checker will experience an increase of the running time, which is proportional to the overall number of collection attempts.

Another difference between a common run-time system and a model checker regards the possibility of unwinding forward moves. When a model checker, like SPIN[7], encounters an already visited state it performs a backward move that matches its last forward move on the trail stack. As result, the model checker restores the current state to one of its previously visited states. For simple transitions, as for example assignments, only a small amount of unwinding information is kept directly on the trail stack. There are however cases in which a state has to be entirely copied on the stack in order to be used later in performing a backward move. The model checker keeps a separate stack for states that have been thoroughly modified by forward moves, denoted as the *state stack*.

When the garbage collector runs in a transition, it might thoroughly modify the heap area along with the adjacent tables i.e., the offset, size and type table. In order to be able to restore the transition's source state, the model checker needs to save the entire heap area along with the additional tables on the state stack. This is needed because it is actually impossible to predict the behavior of the collector i.e., which objects will be reclaimed. As discussed in Section 2, each garbage collection algorithm keeps some additional information about heap-allocated objects e.g., the reference table. As the latter will be modified by a collector run, it also need to be saved on the state stack. This results in a memory overhead that is proportional with the state stack depth.

The following discussion regards implementation details of each collection algorithm in particular. An estimation of the time and space overhead introduced by each algorithm is reported.

3.1 Reference Counting

The reference counting collector keeps in a global table the number of references to each object. As discussed in Section 2, this is the number of pointers that

directly point to the object. The collector runs whenever the reference number has become zero for some objects. These objects are deleted, and then the collector runs recursively for all objects pointed to by the deleted objects fields. The collection stops when no other reachable object can be deleted. The time taken by a collector run is at most $t_{delete} \times N_{reachable}$, where $N_{reachable}$ is the number of objects that are reachable from the set of program pointers, denoted in what follows by `ROOTS`, at the point where the collection was attempted.

An interesting property of the reference counting collection regards the possibility of performing a partial static evaluation of the set `ROOTS`, depending on the nature of the statement that might trigger the collection. For pointer assignment statements, the set `ROOTS` contains only the left-hand side pointer. In this case, garbage collection has to be attempted before the assignment takes place. For statements that exit from functions and local scopes, the set `ROOTS` contains all the local pointers. The latter situation involves explicit delete statements. In this case, the set `ROOTS` contains all pointer fields of the deleted object. As said before, the type of an object cannot be statically evaluated, this information being encoded at run-time into the type table. Depending on the given object type, the collector identifies its layout that is, the position of its pointer fields.

In order to estimate the time overhead introduced by the reference counting collection, let us denote by C_{rc} the total number of collector runs. This is the total number of successful collections, which is always less than the total number of the garbage collector invocations, because the reference counting collector stops as soon as no object can be deleted. With the above notations, the worst-case time overhead introduced by the reference counting collection is $t_{delete} \times N_{reachable} \times C_{rc}$. The number $N_{reachable}$ depends on the way program data is structured. Moreover, it may differ from one collection to the next one. In order to be able to evaluate the time of a collection, in what follows we assume that $N_{reachable}$ represents an average value.

The space overhead can be estimated considering that every collection requires the saving of the type and reference table on the state stack. In practice, the size of a type table entry is 4 bytes (integer), while the reference table entry can be represented on 2 bytes (short). Consequently, each collection introduces a space overhead of $6 \times N_{max}$ bytes, where N_{max} is the maximum number of heap objects. If C_{rc} represents the overall number of reference counting collections performed during model checking, each one needing the save of tables on the state stack, the space overhead is $6 \times N_{max} \times \log(C_{rc})$. We have approximated here the maximum depth of the state stack to $\log(C_{rc})$. In practice, the time and space overheads show to be quite small, therefore we have set reference counting to be the default garbage collection mode in dSPIN.

3.2 Mark and Sweep

The mark and sweep collection performs a thorough scan of the reachable data and marks all objects it encounters during the scan. As mentioned in Section 2, the mark phase starts with the set of all live program pointers `ROOTS`. This cannot be anymore statically evaluated, because the stack layout of every process cannot

be statically determined. This problem can be overcome by giving a description of the global pointers and also the local pointers declared in each proctype or function. Such a data structure is called a *pointer map* and is built by the verifier generator.

To find all the roots, the collector starts at the top of each process stack and scans downward, frame by frame. Each frame keys the function or proctype that corresponds to the next frame, giving the entry into the pointer map. In each frame, the collector marks starting from the pointers in that frame and, in the end, it marks starting from the global pointers. The mark phase is recursive, the spanning being done according to the object types, which are kept into the type table.

When the mark phase completes, the sweep phase deletes all unreachable data. In order to do that, the entire heap area must be scanned every time the collector runs. Let us denote by C_{ms} the total number of collector runs. This is the total number of collector invocations, because the mark and sweep collector always performs a complete scan of the reachable data every time it is invoked. With the above notations, the worst-case time overhead introduced by a mark and sweep collection is $(t_{mark} \times N_{reachable} + t_{delete} \times (N_{max} - N_{reachable})) \times C_{ms}$.

The space overhead is introduced also by the need to save the type table on the state stack. As previously explained, the mark table is written by the mark phase and cleared each time by the sweep phase, therefore we don't need to save it on the state stack. The size of the type table entry is 4 bytes. Consequently, the space overhead introduced by a mark and sweep collection is $4 \times N_{max}$ bytes, where N_{max} is the maximum number of objects. The state is saved on the stack each time the collection was successful that is, at least one object has been deleted. We denote this number by C_{ms}^{del}, therefore the overall mark and sweep space overhead is $4 \times N_{max} \times \log(C_{ms}^{del})$.

3.3 Overhead Comparison

Given the evaluations for time and space overheads previously introduced, we attempt to find out under which circumstances reference counting collection is better than mark and sweep or vice-versa. We remind the reader that all evaluations are related to our implementation of these algorithms in dSPIN.

We denote in what follows, for reference counting, the overall time overhead by $t_{rc} = t_{delete} \times N_{reachable} \times C_{rc}$ and space overhead by $s_{rc} = 6 \times N_{max} \times \log(C_{rc})$. For mark and sweep we denote the overall time overhead by $t_{ms} = (t_{mark} \times N_{reachable} + t_{delete} \times (N_{max} - N_{reachable})) \times C_{ms}$ and space overhead by $s_{ms} = 4 \times N_{max} \times \log(C_{ms}^{del})$.

The time overhead comparison is given by:

$$\frac{t_{ms}}{t_{rc}} = \frac{C_{ms}}{C_{rc}} \times \left(\frac{t_{mark}}{t_{delete}} + \frac{N_{max}}{N_{reachable}} - 1 \right) \tag{1}$$

The first term C_{ms}/C_{rc} is always greater than one, because mark and sweep runs every time it is invoked, while reference counting stops when no objects

can be deleted. For the second term of the expression we can give a worst-case evaluation. More precisely, it is greater than one if $N_{max} \geq 2 \times N_{reachable}$. This gives us a sufficient condition for reference counting collection to run faster than mark and sweep. We have found that, in practice this condition is usually met by common programs.

The space overhead comparison is given by:

$$\frac{s_{rc}}{s_{ms}} = 1.5 \times \frac{\log(C_{rc})}{\log(C_{ms}^{del})} \tag{2}$$

The second term of the expression depends on the program data layout. If the program does not use cyclic data structures then both collection algorithms reclaim almost the same number of objects that is, $C_{rc} \approx C_{ms}^{del}$ and consequently, $s_{rc}/s_{ms} \approx 1.5$. As will be reported in what follows, in practice the space overhead introduced by mark and sweep is always smaller than the one introduced by reference counting.

4 Experimental Work

We have carried out a number of experiments in order to asses the practical value of garbage collection used in combination with model checking. Obviously, an immediate advantage is tracking down programs that may create an unbounded number of objects. Moreover, using garbage collection reduces the number of states in verification and consequently, the model checker time and space requirements.

The first part of this section reports experiments that are performed on the dSPIN specification of a B-tree structure. The reduction in number of states is compared to the time and space overhead introduced by both collection algorithms. The remainder of this section is concerned with an estimation of the best and worst case complexity in a doubly linked list example. All analysis time reports are obtained from the Unix `time` command on a 256 Mb RAM UltraSparc 30 at 300MHz workstation. Small times (under 0.5 seconds) tend to be inaccurate because of the operating system overhead.

4.1 The B-tree Example Revisited

The example considered here is a revision of the test case reported in [5]. The system describes a B-tree structure that is accessed concurrently by two updater processes. A mutual exclusion protocol is used in order to ensure the data consistency while allowing for simultaneous updater accesses to the structure. In order to avoid an unbounded growth of the structure, each updater stops its execution when a predefined maximum depth is reached. The specification was tested for absence of deadlocks. The example is parameterized with respect to the following variables:

- K denotes the B-tree order,

– D denotes the maximum B-tree depth,

In the first case we have explicit delete statements inserted into the model. Table 1 shows the results obtained by performing analysis in presence of explicit delete statements, first without garbage collection, then applying the two collection algorithms, i.e., reference counting (RC) and mark and sweep (MS), that are implemented in dSPIN.

Table 1. Explicit Delete B-tree

K,D	States	Memory (Mb)	GC (runs)	Time (h:m:s)	Options
4, 2	4620	3.345	-	0:0:1.0	-
	4620	3.895	RC (0)	0:0:1.4	-
	4620	3.712	MS (1591)	0:0:1.2	-
2, 3	440486	58.062	-	0:1:41.0	COLLAPSE
	313316	42.858	RC (124)	0:1:39.2	COLLAPSE
	237442	31.837	MS (91247)	0:1:08.7	COLLAPSE
2, 4	5.65413e+07	135.772	-	3:33:37.5	BS (9.49519)
	5.63116e+07	136.468	RC (25745)	4:11:10.4	BS (9.53393)
	5.34694e+07	136.242	MS (33558759)	3:50:04.8	BS (10.0408)

The overall number of collector runs is also presented in the table. For reference counting, this is the total number of successful collections that is, collections in which at least one object has been reclaimed. This comes as a consequence of the fact that reference counting collection stops when it cannot reclaim any more objects. Instead, mark and sweep collection performs exactly one sweep of the heap area every time it is invoked. In this case, the number of collector invocations is the number of collector runs.

It is to be noted that garbage collection acts also as a complexity reduction mechanism. The explicit delete statements that have been inserted into the model tend to increase the overall number of states. Our intuition is that semantically-equivalent states do not match because of different orderings of objects into the heap area. As garbage collection constantly reduces the number of objects it also reduces the number of possible interleavings in the heap representation. Recently, the implementors of the Java PathFinder tool [6] have considered the possibility of representing the heap objects in the same way, disregarding their creation order, which makes a further reduction in the number of states.

In the first case (4, 2) garbage collection is inefficient. Allowing for a maximum tree depth of 2 implies that no object is ever deleted during an update. In this case garbage collection introduces only an overhead in memory and space but does not improve the verification performance. The second case (2, 3) shows clearly how garbage collection reduces the model checker state space and conse-

quently, its time and space requirements. Due to the fact that in our model, the B-tree nodes are doubly linked, mark and sweep collection is more efficient than reference counting. The latter case (2, 4) was analyzed using bitstate hashing (BS). Table 1 shows also the hash factor reported by the verifier (following the BS). It is to be noted that, in this case, using garbage collection reduces the state space complexity and improves the hash factor.

The second suite of experiments has been performed by replacing the explicit deletes with `skip` statements. Table 2 shows the results. In general, the number of states experiences a remarkable decrease, because the lack of explicit deletes makes semantically-equivalent states match in most cases.

Table 2. No Explicit Delete B-tree

K,D	States	Memory (Mb)	GC (runs)	Time (h:m:s)	Options
4, 2	637	1.830	-	0:0:0.1	-
	637	1.920	RC (18)	0:0:0.1	-
	637	1.877	MS (161)	0:0:0.1	-
2, 3	1257	2.497	-	0:0:0.1	-
	1257	2.587	RC (14)	0:0:0.1	-
	1257	2.429	MS (325)	0:0:0.1	-
4, 3	3025	4.935	-	0:0:0.3	-
	3025	4.848	RC (20)	0:0:0.3	-
	3025	4.013	MS (739)	0:0:0.3	-
2, 4	23852	11.987	-	0:0:30.3	COLLAPSE, MA
	23852	11.709	RC (14)	0:0:23.5	COLLAPSE, MA
	30346	9.242	MS (7451)	0:0:30.9	COLLAPSE, MA

In this case, using garbage collection does not reduce the state space, rather tends to slightly increase it, as it is with the case (2, 4) verified using mark and sweep. However, it is to be noticed that the memory requirements tend to decrease, as a consequence of the decrease of the current state size. In the case (4, 2), the memory overhead introduced by the garbage collection balances the gains, resulting in a greater memory requirement. But, as discussed, the space overhead introduced by garbage collection tends to increase logarithmically with the overall number of collections, and therefore with the total number of states. The following cases (2, 3), (4, 3) and (2, 4) show an actual decrease of memory taken up by the verifier when using garbage collection.

4.2 Worst and Best Case Estimation

We have performed a number of experiments in order to give an upper and lower bound to the efficiency of the collection algorithms. The test case is a specification of a doubly linked list which is updated sequentially by one process. The

process does a fixed number of list updates, counted up by an integer variable. The purpose of the counter variable is to ensure that the number of states does not change when garbage collection is used. As the value of the counter is incremented at each iteration, the states from the current iteration will not match the states from previous iterations. The number of states is used as a reference point in our evaluation.

In the worst case, a fixed number of cons cells is inserted into the list. All cells are continuously referenced by the head of the list during program runtime, which makes garbage collection impossible. In the best case, the same number of cons cells is inserted into the list, but after each insertion, every cell is then extracted by resetting the list reference to it. No explicit deletes are performed, the cells being reclaimed only by the garbage collector. Table 3 shows the analysis results.

Table 3. Worst and Best Case Garbage Collection

GC	Runs	States	Memory (Mb)	Time (secs)
i. Worst Case				
-	-	12242	253.009	20.1
RC	0	12242	295.154	20.9
MS	6119	12242	281.105	26.3
ii. Best Case				
-	-	16832	386.079	50.9
RC	1530	16832	231.66	17.3
MS	7650	16832	208.252	8.5

It is to be noticed that the worst-case space overhead introduced by reference counting is greater than the one introduced by mark and sweep. On the other hand, the worst-case time taken up by the reference counting is smaller than the one taken up by mark and sweep. In this case both collection algorithms are inefficient because they cannot reclaim any object. In the best case, mark and sweep yields much better results in both space and time than reference counting, making the analysis more efficient. This comes as a consequence of the fact that, in this case, the rate of successful mark and sweep collections is rather big i.e., at least one occurs in each iteration.

5 Conclusions

Providing model checkers like dSPIN[1] with support for garbage collection allows analysis of a wider class of real-life software systems. This occurs as a

[1] The dSPIN source code is distributed via the URL: http://www.dai-arc.polito.it/dai-arc/auto/tools/tool7.shtml

consequence of the widespread use of garbage collection in current programming environments. In particular, programs that rely strictly on garbage collection, like the ones written in Java, tend to create untractable models because of the unbounded growth of their working memory. Despite a limited run-time overhead introduced by the collection algorithms, embedding them into model checking seems to improve the analysis results in general, acting also as a state reduction mechanism.

When implementing such algorithms in a model checker, attention has to be paid to the particularities of the model checking environment, such as the possible backward moves. An evaluation of the time and space overhead introduced by garbage collection has been given. Although such an evaluation is tightly related to the layout of the data structures used in the program, worst and best case estimation can be given. Experiments have been carried out in order to assess the practical value of our work.

References

1. Andrew W. Appel: Modern Compiler Implementation in Java. Cambridge University Press, (1998)
2. J. Corbett: Constructing Compact Models of Concurrent Java Programs. Proc. of International Symposium on Software Testing and Analysis (1998)
3. J. Corbett, M. B. Dwyer et al.: Bandera: Extracting Finite-state Models from Java Source Code. Proc. 22nd International Conference on Software Engineering (2000) 439–448
4. C. Demartini, R. Iosif, R. Sisto: A deadlock detection tool for concurrent Java programs. Software: Practice & Experience, Vol 29, No 7 (1999) 577–603
5. C. Demartini, R. Iosif, R. Sisto: dSPIN: A Dynamic Extension of SPIN. Lecture Notes in Computer Science, Vol. 1680, Springer-Verlag, Berlin Heidelberg New York (1999) 261–276
6. G. Brat, K. Havelund, S. Park, W. Visser: Java PathFinder: Second Generation of a Java Model Checker. Workshop on Advances in Verification (2000)
7. G. Holzmann: The Design and Validation of Computer Protocols. Prentice Hall, (1991).
8. R. Iosif: The dSPIN User Manual. http://www.dai-arc.polito.it/dai-arc/auto/tools/tool7.shtml