Exploiting Heap Symmetries in Explicit-State Model Checking of Software

Radu Iosif

Computer and Information Sciences Department
318 Nichols Hall, Kansas State University
Manhattan, KS 66506, USA
iosif@cis.ksu.edu

Abstract

Detecting symmetries in the structure of systems is a well known technique falling in the class of bisimulation (strongly) preserving state space reductions. Previous work in applying symmetries to aid model checking focuses mainly on process topologies and user specified data types. We applied the symmetry framework to model checking object-based programs that manipulate dynamically created objects, and developed a linear-time heuristic for finding the canonical representative of a symmetry equivalence class. The strategy was implemented in the object-based model checker dSPIN and some experiments, yielding encouraging results, have been carried out.

KEYWORDS: symmetry reductions, property preservation, software model checking, object-based programs

1 Introduction

Finite-state verification techniques, such as model checking [2], exhaustively check a system in order to decide whether it satisfies a specification. Such specifications are usually written in some temporal logic, such as CTL [4] or LTL [20] and the verification procedure is completely automatic. These methods have proven effective in reasoning about hardware components and protocol specifications. Recent advances in tool support have allowed these techniques to be applied to software systems, written in high-level languages such as C [12] and Java [5]. A major obstacle to the transfer of finite-state verification technology to software is the "state explosion problem": the size of a finite-state model increases exponentially with the number of its components, as different interleavings of concurrent threads generate different states. This increase seems to be even more extreme in case of software. One reason for this is that the number of program components may vary along its execution: new threads and objects are created at run-time. This causes the interleaving factor of the program experience a rapid growth. The focus of this paper is on dynamic systems, in which the number of components may constantly change along its executions.

Work has been done to implement cost-efficient software verification toolsets, including source-code model extractors [5] and software-oriented model checkers [15, 21]. The former class of tools relies on applying abstract interpretation [6] in order to curb the state space explosion. Specific to the latter category are on-the-fly optimizations such as partial-order reductions [9] and symmetry reductions [3]. Attempting to apply model checking to software written in modern programming languages opens a number of problems related to the run-time complexity of such programs. Verifiers using explicit state representations [15, 21] have shown the ability of efficiently modeling run-time mechanisms such as: dynamic object creation, virtual method dispatch and garbage collection [15, 16, 19].

A central issue in state space reduction is the question of property preservation. Using temporal logic as specification language divides state-space reduction techniques into: "simulation" (weakly) preserving and "bisimulation" (strongly) preserving transformations. Simulation preserving reductions (e.g., abstract interpretation) are capable of verifying properties only, while the bisimulation preserving ones

can be used for both verifying and refuting properties (i.e., report upon both satisfaction and failure). Practical results [5] show that weakly preserving techniques achieve better reductions in expense of some precision losses. That is, errors could be reported even when they do not exist in the real system, and additional refinements are required to eliminate such false negatives. On the other hand, strongly preserving reductions do not induce false errors, therefore verification results need not be subsequently refined.

In this paper we investigate how to exploit symmetries induced by data structures (objects) to reduce the complexity of the model checking problem applied to software. Normally, high-level programming languages, such as Java, do not allow programmers to observe the locations of objects in memory; a reference is an *abstract* location. Thus, the semantics of a program's execution does not depend on the different orderings of objects in memory. Since explicit-state model checkers such as dSPIN [15] and JPF [21] use explicit state encodings, states of a program that differ only in the positions of objects in the heap will cause the model checker to store states that are observationally equivalent. Since such states are bisimilar, they cannot be distinguished by properties written in temporal logics. Structural symmetries induce an equivalence relation on states. In practice, detecting symmetries has been shown to produce a stronger approximation of bisimulation. Unfortunately, the general problem of computing symmetry equivalence classes [3] (the orbit problem) has no polynomial-time solution. In general this makes it too costly to perform during model checking.

Our approach to symmetry reductions is novel in that it exploits the shape of the heap configuration to achieve the optimal reduction for the entire class of heap symmetries. The algorithm works for programs handling variable numbers of objects, its complexity being linear in the number of objects present in a state. The key for the correctness of our heuristic is that all program variables, including object fields, can be ordered at compile-time. This ordering induces a total ordering of objects based on their "reachability traces", i.e., chains of variables that reach non-garbage objects. The sorting of traces induces a permutation on objects and applying such a permutation to the current state gives the unique representative of the symmetry equivalence class. A state search algorithm will store only such representatives in the state space, thus reducing the memory requirements of a model checker. Since our algorithm determines all reachable objects in a given program state, it can be combined with on-the-fly garbage collection, which is another technique already used by existing software model checkers [16, 19].

The main contribution of this paper is the presentation of a new symmetry reduction strategy that we have implemented on top of the dSPIN model checker [15]. This is an extension of the well-known model checker SPIN [13], providing effective support for the finite-state verification of object-based programs. The technique has been evaluated on two non-trivial test cases involving dynamic data structures shared between multiple concurrent threads. Preliminary results show reductions in space (number of stored states) ranging from one to four orders of magnitude. Even though the time required to compute transitions is increased by a stable factor (approximately 1.4), important savings in memory determine overall reductions in verification time. For instance, a verification problem originally requiring over 1 Gbytes can be done now in 12 Mbytes with a reduction in verification time from 85 minutes to 10 seconds.

1.1 Related Work

Among the first to use symmetries in model checking were Clarke, Filkorn and Jha [3], Emerson and Sistla [8] and Ip and Dill [17]. These approaches consider systems composed of a fixed number of active components (processors) [3], variables of a special symmetry-preserving data type (scalarset) [17] as well as symmetries of specifications [8]. Experimental results show reductions of memory requirements, that are in most cases exponential in the number of replicated system components. Initially, the test cases were hardware cache consistency protocols. The more recent work of Godefroid [10] applies symmetry reductions to stateless model checking of programs.

Reducing the complexity of representatives computations has been tackled by the work of Bosnacki and Dams [7]. They describe a number of heuristic strategies that use sorting of state components (processes) to aid the computation of orbit representatives. The idea of using sorting permutations first occurs here, even though the sorting is based on the contents of the components, rather than their organization within the program state. The techniques proposed are shown to be efficient when the number of components with identical contents is low. Indeed, this is the case when the components in question are processes, but this might not hold when symmetries of pure data components (objects) are considered, since a program can create and store an arbitrary number of identical objects.

The problem of exploiting heap symmetries in software model checking has been informally addressed by Visser and Lerda in [19]. To our knowledge, they are the only other group that tackled this issue so

far. They solve the ordering problem by storing with each allocator statement the memory location it returns when it is first executed. The first-time location will be returned by all subsequent calls to the allocator. In this way, the same partial ordering of objects is maintained on different execution traces. This approach looks attractive due to its simplicity, still no formal evidence of its optimality has yet been provided by the authors.

The rest of this paper is organized as follows: Section 2 informally describes the ideas behind symmetry reductions using an example program written in Java, Section 3 introduces the symmetry framework that is to be used in the rest of the paper, Section 4 describes the reduction strategy, giving a formal proof of its correctness. Section 5 describes the implementation of our technique, reporting also some experimental results, and last, Section 6 contains conclusions and a description of future work.

2 An Example

This section presents an example program for which detection of heap symmetries can be used to reduce the number of states. To improve readability, the example is written in Java, but one can easily cast it in a different object-oriented language.

The program fragment in Figure 1 illustrates the implementation of a message dispatcher ensuring communication between an arbitrary number of clients and servers. Messages are instances of class `Message`, containing priority numbers, as shown in Figure 1 (b). Such messages are produced by client threads, shown in Figure 1 (b), and sent to the `MessageQueue` in Figure 1 (a) using its `send` method. The messages are stored in a priority queue in the ascending number of their priority numbers. We will not concentrate on the details of the clients, assuming that priority numbers are the result of some internal computation. We focus on the following issue: due to the concurrency involved, messages are inserted in the request queue in a fixed order that does not always match the order in which these objects have been created. As a consequence, the representation of the message queue in memory differs between scheduling scenarios, even though the semantics of the program computations remains the same.

```
class MessageQueue {
  Message head = new Message(0);
  Message tail = head;

  synchronized void send(Message m) {
    Message curr = tail;
    Message last = tail;
    while (curr!=null && curr.prio>m.prio) {
      last = curr;
      curr = curr.next;
    }
    if (curr==null)
      head = m;
    last.next = m;
    m.next = curr;
  }
}
```

(a)

```
class Message {
  int prio;
  Message next;
  Message(int p) { prio = p; }
}

class Client extends Thread {
  MessageQueue q;
  int p;
  public void run() {
      ...
    Message m = new Message(p);
    q.send(m);
      ...
  }
}
```

(b)

Figure 1: Message Queue Example

Consider a program in which two `Client` threads concurrently create two messages with priorities 1 and 2, respectively and send them to a shared message queue. We describe two possible interleavings of these threads assuming that the program uses a first-free allocator, i.e., the first memory location not already allocated will always be returned by the allocator. A general abstraction of the underlying memory area is given in terms of a discrete, total ordered set of locations denoted by l_0, l_1, l_2, etc. Figure 2 shows two possible configurations of the heap, in which messages are represented by boxes, labeled with priority numbers, and references are depicted as arrows labeled with field identifiers. The locations where the objects reside in memory are also represented. The `MessageQueue` class is initialized by storing

in a dummy message having priority 0. One scenario considers that the first client thread creates a message at memory location l_1, setting its priority to 1 and sends it to the queue. The second thread then proceeds by allocating another message at location l_2, setting its priority to 2 and then sending it to the queue. The resulting heap configuration is shown in Figure 2 (a). In the second scenario, we have the second client proceed first, allocate a message at location l_1 with the priority number 2, and send it to the message queue. The first client will then proceed with the creation of a message and since the next available location is l_2, it will be returned by the allocator, and a message with priority 1 will be created at l_2. As messages are queued following the order of their priorities, the second created message will be inserted before the first one and the resulting heap configuration will be the one in Figure 2 (b).

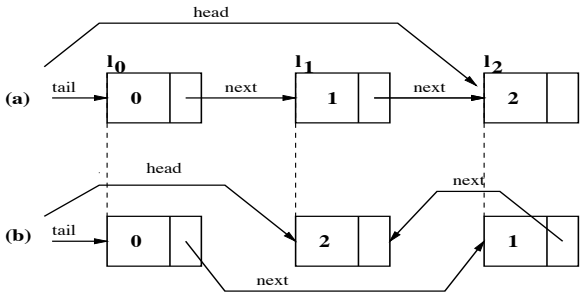

Figure 2: Message Queue Configurations

Both program configurations are equivalent since the position of objects in memory does not affect the behavior of the message queue. Nevertheless, an explicit-state model checker has no way to detect this fact, since it compares the states according to the values stored in memory.

3 Background

In this section we present a slight variation of the symmetry group framework, first introduced in [3, 8]. The classical framework is based on the notion of group automorphism and deals in principle with systems whose states have a constant number of components. In our case, the number of components may vary from state to state. A permutation π is a bijection from the set of natural numbers $\{0, \ldots, n-1\}$ to itself. Let G_n be the group of all such permutations with function composition and inverse operation.

Let $\mathcal{P}$ be a set of atomic propositions over states. The state space of a (concurrent) program can be represented by a labeled transition system (LTS) $M = \langle \Sigma, S, \rho, L, \eta \rangle$ where:

- Σ is a finite set of labels also known as the "alphabet",
- S is a set of program states,
- $\rho : S \times \Sigma \to 2^S$ is the transition function,
- $L : s \to 2^{\mathcal{P}}$ is a function mapping states to sets of atomic propositions that hold in those states,
- $\eta : s \to \mathbb{N}$ gives the number of components that are present in each state.

Optionally, we write $s \xrightarrow{t} s'$ to denote that $s' \in \rho(s, t)$ for some $s, s' \in S$ and $t \in \Sigma$. The permutations on numbers can be extended in general to permutations on states. Let $\pi(s)$ denote the application of permutation π to state s. The actual definition of such applications depends on the structure of the states and will be defined later, for a particular class of systems.

Definition 1 *Let $M = \langle \Sigma, S, \rho, L, \eta \rangle$. A binary relation $\equiv \subseteq S \times S$ is a symmetry iff, for all $s \equiv t$ all the following hold:*

- $L(s) = L(t)$,
- $\eta(s) = \eta(t)$,
- $\pi(s) = t$ *for some* $\pi \in G_{\eta(s)}$.

It is easy to prove, from the properties of a group, that $\equiv$ is actually an equivalence relation. The equivalence class, also known as the *orbit* of a state s is usually denoted by $[s]$.

Definition 2 *Given an LTS $M = \langle \Sigma, S, \rho, L, \eta \rangle$ and a symmetry relation $\equiv$ on S, the quotient transition system for M modulo $\equiv$ is an LTS $M_{\equiv} = \langle \Sigma, S', \rho', L', \eta' \rangle$, where:*

- $S' = \{[s] \mid s \in S\}$,
- $\rho'([s], t) = \{[s''] \mid \exists\, s'.\ s' \equiv s \ \wedge\ s'' \in \rho(s', t)\}$, *for all* $s \in S$ *and* $t \in \Sigma$,
- $L'([s]) = L(s)$, *for all* $s \in S$,
- $\eta'([s]) = \eta(s)$, *for all* $s \in S$.

The states of a quotient transition system are equivalence classes of states from the original transition system and a transition occurs between equivalence classes if and only if a transition with the same label occurs between states from the non-reduced transition system. It is clear, from the first two points of definition 1, that L' and η' are well-defined on S'. As a state in $M_{\equiv}$ contains possibly more than one state from M, it is potentially more efficient to model check the former LTS, provided that they always generate equivalent computations.

Definition 3 *Let $M_1 = \langle \Sigma, S_1, \rho_1, L_1 \rangle$ and $M_2 = \langle \Sigma, S_2, \rho_2, L_2 \rangle$. A binary relation $\approx \subseteq S_1 \times S_2$ is a bisimulation iff, for all $s_1 \approx s_2$, both the following hold:*

- *for all $s_1' \in S_1$ such that $s_1 \xrightarrow{t} s_1'$, there exists $s_2' \in S_2$ such that $s_2 \xrightarrow{t} s_2'$ and $s_1' \approx s_2'$,*
- *for all $s_2' \in S_2$ such that $s_2 \xrightarrow{t} s_2'$, there exists $s_1' \in S_1$ such that $s_1 \xrightarrow{t} s_1'$ and $s_1' \approx s_2'$,*
- $L(s_1) = L(s_2)$.

If $\approx$ is total, one can say that M_1 and M_2 are bisimilar (i.e., $M_1 \approx M_2$). It is known fact that bisimilar states cannot be distinguished by formulas of mu-calculus [18] or any of its sub-logics, such as computation-tree logic (CTL) [4] or linear-time temporal logic (LTL) [20]. The following theorem has been proved in [3, 8, 17, 14]:

Theorem 1 *Given $M = \langle \Sigma, S, \rho, s_0, L, \eta \rangle$, for all $s, t \in S$, if $s \equiv t$ then $s \approx t$.*

A direct consequence is that, given an LTS M, its quotient modulo symmetry equivalence $M_{\equiv}$ is related with it by a bisimulation, and since $M_{\equiv}$ is smaller, it can be used in model checking, instead of the original M. The barrier to apply this strategy in practice is related to the computation of the quotient transition system $M_{\equiv}$ from the original M. An efficient implementation needs to build $M_{\equiv}$ on-the-fly, without having the explicit structure of M. Unfortunately, the general problem of finding the equivalence class of a state (the *orbit problem*) is known to be as hard as proving graph isomorphism, for which no polynomial-time solution is known to exist [3]. Solutions proposed in the literature either deal with incomplete equivalence classes for which the orbit problem has polynomial solution [3] (i.e., the *bounded orbit problem*), or use heuristic search strategies [7].

4 Symmetry Reductions

This section focuses on formal aspects of heap symmetries. We start introducing a model for program states in order to define the meaning of permutations on states. Next, we describe our symmetry reduction strategy and give formal reasons for which it is optimal.

As we deal with software model checking, it is important to consider software written in common programming languages that provide support for dynamically allocated objects. We introduce the concept of *object-based* programs as a model for a wide class of programs that manipulate a finite number of objects at a time, by dynamically creating such objects, linking them into the reachable objects graph and modifying their internal state. The operational semantics of a language for describing object-based programs has been defined in [14].

Consider the semantic domains in Figure 3. A program configuration s_P is a triple s, h, n whose first element is a store, i.e., a partial mapping from variable identifiers to values, second element is a heap, partially mapping memory locations to stores and third element is the number of components in that configuration, i.e., $\eta(s_P) = n$. In what follows, we will refer to the stores in the image of h as "objects". In general, we use the bottom ($\perp$) symbol to represent undefinedness of partial mappings on some elements from their domains. The notation $S_\perp$ where S is a set, actually stands for $S \cup \{\perp\}$. For the sake of simplicity, consider that each variable in the program, denoted by $v \in \mathit{Variables}$ is a reference variable, i.e., its values can only be memory locations. Formally, the set of all memory locations has been denoted by *Locations* in Figure 3. This set is considered finite that is, $\mathit{Locations} = \{l_0, l_1, \ldots, l_{n-1}\}$

$$
\begin{array}{rcl}
l & \in & \mathit{Locations} \\
s & \in & \mathit{Stores} = \mathit{Variables} \longmapsto \mathit{Locations}_{\perp} \\
h & \in & \mathit{Heaps} = \mathit{Locations} \longmapsto \mathit{Stores}_{\perp} \\
s_P & \in & \mathit{States} = \mathit{Stores} \times \mathit{Heaps} \times \mathbb{N}
\end{array}
$$

Figure 3: Semantic domains

and, for all $l_i \in \mathit{Locations}$ and $\pi \in G_n$, let $\pi(l_i) = l_{\pi(i)}$ and $\pi(\perp) = \perp$. With these considerations, we define the meaning of a permutation π on state a $s_P = s, h, n$ below:

$$
\begin{array}{rcll}
\pi(s_P) & = & \pi(s), \pi(h), n & (1) \\
\pi(s) & = & \lambda x.\pi(s(x)) & (2) \\
\pi(h) & = & \lambda l.\lambda x.\pi(h(\pi^{-1}(l), x)) & (3)
\end{array}
$$

Informally, the equations above say that, applying a permutation to a state, will permute all locations that are values of reference variables in the store and in each heap object. The objects in the heap are also permuted, by the inverse permutation, in order to consistently reflect this change.

As an example, let us apply the permutation $\{(0,0),(1,2),(2,1)\}$ to the state in Figure 2 (b). The store variable *head* has the value $\pi(l_1) = l_2$ in the permuted state, while *tail* is not changed by π ($\pi(l_0) = l_0$). The value of the *next* field of the object at l_0 becomes $\pi(l_2) = l_1$ and the *next* field of the object at l_2 becomes $\pi(l_1) = l_2$. Second, the objects at l_1 and l_2 are swapped by the inverse permutation ($\pi^{-1} = \pi$, in this case) and the permuted state is the one in Figure 2 (a).

Since explicit-state model checkers for software [15, 21] have an explicit representation of the program heap by keeping track of object locations in memory, the state space search algorithm can be optimized by computing, for each reachable state, its symmetry equivalence class. We can conclude that the search has reached a visited state when at least one of the states in the orbit has been already visited by the state space search. A generic symmetry reduced state search algorithm has been described in [7]. The algorithm applies a given function to compute, for each state, the representative of its equivalence class and stores only such representatives in the state space. A representative function is called *canonical* if, for each state in the orbit, it chooses the same representative state, yielding the best case reduction.

Our approach to symmetry reductions exploits the *shape* of the heap in order to find the canonical representative of a symmetry equivalence class. Consider that a program state is viewed as a directed graph where objects are nodes and reference variables relating two objects are edges. We complete this view introducing, for each store variable a special "entry" node. Instead of sorting the state on lexicographical basis, we can now sort the graph topologically, attempting to partition it using depth-first spanning trees (DFST). A DFST [11] is a tree rooted at one of the entry nodes, trying to cover the longest possible paths in the graph.

Consider the set of reference variables $\mathit{Variables}$ in Figure 3 and $\preceq_V \subseteq \mathit{Variables} \times \mathit{Variables}$ a total ordering of all reference variables declared in a program. The existence of such ordering is a key issue for the correctness of our strategy. It is guaranteed that the ordering can be found at compile-time, and one might consider for instance alphabetical order, declaration order, etc. This ordering entails a natural ordering on sequences of variables, denoted by $\preceq_V^* \subseteq \mathit{Variables}^* \times \mathit{Variables}^*$. The notation min_V^* denotes the greatest lower bound with respect to $\preceq_V^*$ and $\circ$ is sequence concatenation. As a convention, we use the literals v, u to denote sequences of variables and x, y to denote variables. Consider the following mappings:

$$
\begin{array}{rcl}
\mathit{reach} & : & \mathit{States} \times \mathit{Variables}^* \rightarrow \mathit{Locations}_{\perp} \\
\mathit{reach}(s_P, v) & = & \left\{ \begin{array}{ll} s(x) & \text{if } v = x \\ h(\mathit{reach}(s_P, u), x) & \text{if } \mathit{reach}(s_P, u) \neq \perp \text{ and } v = u \circ x \\ \perp & \text{otherwise} \end{array} \right.
\end{array}
$$

and

$$
\begin{array}{rcl}
\mathit{trace} & : & \mathit{States} \times \mathit{Locations} \rightarrow \mathit{Variables}_{\perp}^* \\
\mathit{trace}(s_P, l) & = & \left\{ \begin{array}{ll} min_V^*\{v \mid \mathit{reach}(s_P, v) = l\} & \text{if } \mathit{reach}(s_P, v) = l \text{ for some } v \in \mathit{Variables}^* \\ \perp & \text{otherwise} \end{array} \right.
\end{array}
$$

Informally, $reach(s_P, v)$ returns the location that is reachable in a state s_P by following the chain of reference variables given by the sequence v. Since some sequences might denote undefined locations (dangling chains), *reach* is allowed to return $\perp$ to express undefinedness. Conversely, $trace(s_P, l)$ returns the least sequence of reference variables that reaches l in state s_P or $\perp$ if no such sequence exists. Next, consider that the sorting of reachability traces induces a permutation, defined as follows:

Definition 4 *A permutation $\pi_{sort} \in G_{\eta(s_P)}$ is said to be "sorting" for s_P iff for all $0 \leq i, j < \eta(s_P)$ such that $trace(s_P, l_i) \neq \perp$ and $trace(s_P, l_j) \neq \perp$, the following holds:*

$$\pi_{sort}(i) \leq \pi_{sort}(j) \iff trace(s_P, l_i) \preceq_V^* trace(s_P, l_j) \tag{4}$$

Let us turn back to the original problem of finding the representatives of symmetry equivalence classes. The discussion concerning sorting permutations was meant to prove that a sorting permutation gives the canonical representative of a symmetry equivalence class. The proof of the following theorem can be found in [14].

Theorem 2 *If $s, t \in States$ are two program states, π_s and π_t are sorting permutations for s and t respectively, then:*

$$s \equiv t \iff \pi_s(s) = \pi_t(t)$$

As an example, consider again the two heap configurations shown in Figure 1, with the following ordering on variables: $tail \preceq_V next \preceq_V head$. For the state in Figure 1 (a), the sorting permutation is the identity, since $trace(l_0) = tail$, $trace(l_1) = tail, next$ and $trace(l_2) = tail, next, next$, therefore $trace(l_0) \preceq_V^* trace(l_1) \preceq_V^* trace(l_2)$. Following theorem 2, the (a) configuration is the representative of its heap symmetry orbit. In the second case (b), since $trace(l_0) \preceq_V^* trace(l_2) \preceq_V^* trace(l_1)$, the sorting permutation is $\pi_{sort} = \{(0,0), (1,2), (2,1)\}$. The result of applying π_{sort} to the heap in Figure 1 (b) matches the state in Figure 1 (a).

Under the simplifying assumption that every location is reachable by a chain of program variables, we claim that the algorithm in Figure 4 generates a sorting permutation for a state. As will be discussed in what follows, this restriction can be lifted. Let us assume that the function $ordered : Stores \rightarrow Variables^*$ returns, for a given store, the $\preceq_V$-ordered sequence of reference variables that have defined values in that store.

The DFST procedure visits all reachable locations in the program state, assigning each location its corresponding depth-first order number (line 7). When a recursive call to DFST visits a location l, the values held by the local variable x along the sequence of calls form a sequence of reference variables v such that $reach(s_P, v) = l$. Moreover, we claim that v is actually the $\preceq_V^*$-minimum of all such sequences, since at line 1, x is assigned to the next variable returned by *ordered* with respect to $\preceq_V$. This property of *ordered* is in fact the key for the correctness of our algorithm. Since v is the least sequence that reaches l in s_P, we have $v = trace(s_P, l)$. Next, if a reachable location l_i has been visited by DFST before l_j, it is the case that $v_1 \preceq_V^* v_2$, where $v_1 = trace(s_P, l_i)$ and $v_2 = trace(s_P, l_j)$. The reason can be found considering the two possible situations:

1. if DFST chooses to follow v_1 before v_2 (line 9), the choice is made (line 1) by the *ordered* function. Then v_1 and v_2 have a common (possibly empty) prefix. Let x_1 (x_2) be the value of x chosen at line 1 in the first (second) case. Consequently, $x_1 \preceq_V x_2$ therefore $v_1 \preceq_V^* v_2$.
2. a call to DFST reaches l_i before a recursive call (line 9) will eventually reach l_j. In this case v_1 is a prefix for v_2.

Let us denote by k_1 (k_2) the value of variable k when a call DFST visits l_i (l_j). Following that k is continuously incremented (line 8) and from the way π_{sort} has been built (line 7) it is the case that $k_1 \leq k_2$, and $\pi_{sort}(i) \leq \pi_{sort}(j)$ follows. Since all locations were supposed to be reachable, DFST will visit all of them, so the resulting π_{sort} is a permutation. Moreover, it respects the condition (4) from definition 4.

The sorting permutation algorithm works in principle only in states where all locations are reachable or, in other words, in those states that do not contain garbage objects. When a state contains garbage, the output of the algorithm in Figure 4 might not be a permutation. This restricition can be easily lifted in practice. As the algorithm marks all reachable locations (line 6) in a state, it can be used also to perform on-the-fly garbage collection during model checking [16]. The solution we use consists in freeing all unmarked (garbage) locations and reindexing the remaining reachable locations. Consider for example the configuration in Figure 1 (b) in which the indexes of all locations occur increased by one and l_0 has become garbage. The algorithm in Figure 4 running on this configuration with the ordering

```
Input: a program state s_P = s, h, n
Output: a sorting permutation π_sort ∈ G_n

DFST(store)
1 for the next x from ordered(store)
2 begin
3       l_i := store(x)
4       if mark[l_i] = unvisited
5       begin
6               mark[l_i] := visited
7               π_sort := [i → k]π_sort
8               k := k + 1
9               DFST(h(l))
10      end
11 end
end DFST

12 for each 0 ≤ i ≤ n − 1
13 begin
14        mark[l_i] := unvisited
15 end
16 π_sort := λx. ⊥
17 k := 0
18 DFST(s)
```

Figure 4: The Sorting Permutation Algorithm

tail $\preceq_V$ *next* $\preceq_V$ *head* outputs the mapping $\pi_{sort} = \{(1,0),(3,1),(2,2)\}$ that is not a functional mapping, since it is undefined in 0. By eliminating the garbage location l_0, all the indexes from the domain of π_{sort} decrease by one, and the result is $\{(0,0),(2,1),(1,2)\}$, which is indeed a sorting permutation.

The worst-case complexity for the algorithm in Figure 4 running in state s is $O(N)$, assuming that N is the number of reachable objects in state s. Let us denote by E the total number of pointers to reachable objects i.e., $E \leq D \times N$, where D denotes the number of reference variables declared in the program. The number of calls to DFST (lines 9, 18) is N since every reachable object is visited only once. The overall number of iterations beginning at line 1 is E, because the statements at lines 3, 4 will be executed in every call to DFST, for every pointer originating in the object passed as parameter to the call. Therefore the algorithm requires at most $O(N + E) = O(N)$ time.

5 Implementation and Experience

We have implemented this reduction strategy in the dSPIN model checker [15]. Similar extensions could be applied to other verification systems that offer explicit support for the representation of a program's heap [21]. We performed experiments involving two test cases: the first one is a model of an ordered list shared between multiple updater threads, while the second models an interlocking protocol used for controlling concurrent access to a shared B-tree structure. Both models are verified for absence of deadlocks, as we performed these preliminary experiments in order to obtain a benchmark of the performances of our technique.

dSPIN is a model checker designed for the verification of software, providing a number of novel features on top of standard SPIN's [13] state space reduction algorithms, e.g., partial-order reduction and state compression. The input language of dSPIN is a dialect of the PROMELA language [13] offering, C-like constructs for allocating and referencing dynamic data structures. On-the-fly garbage collection is also supported [16]. The explicit representation of states allowed the embedding of such capabilities directly into the model checker's core. This showed to be a possible way of bridging the semantic gap between high-level object oriented languages, such as Java or C++, and formal description languages that use abstract representations of systems, such as finite-state automata.

The existence of garbage collection algorithms in dSPIN made the implementation of symmetry reductions particularly easy. We adapted the *mark and sweep* algorithm [16] to generate sorting permutations while visiting the graph of reachable objects. Interestingly enough, there is no need to run the sorting permutation algorithm (Figure 4) more often than the garbage collector. The reason is that the set of statements that might generate heap-symmetric states is a subset of the ones that potentially produce garbage objects.

The first test case models a list ordered by node keys. The list can be updated by an arbitrary number of concurrent processes. There are two processes in our example: an inserter that adds given keys into the list, and an extractor that removes nodes with given keys from the list. Symmetries arise in this case because different interleavings between the inserter and the remover process cause different allocation orderings of nodes with same keys. The example scales in the maximum length of the list (N). The results are shown in Table 1[1]. Although similar with the message queue example from Section 2, the concurrent ordered list example has a finer granularity: accesses to the message queue (Figure 1) are exclusive for the entire structure, while in the ordered list case, more than one updater could access the structure at the same time, yielding more interleavings.Verification of the last case ($N = 10$) without symmetry reductions failed to terminate due to space limitations.

Table 1: Ordered List Example

N	States	Transitions	Speed (states/sec)	Memory (Mb)	Symmetry Reductions
8	766297	935179	20165	200.0	no
	296159	365266	19743	80.5	yes
9	2.29669e+06	2.80801e+06	14817	586.5	no
	727714	899147	18659	163.1	yes
10	4.62012e+06	5.66841e+06	6647	> 1246.1	no
	1.75287e+06	2.16879e+06	16694	370.7	yes

The second example is an interlocking protocol that ensures the consistency of a B-tree* data structure accessed concurrently by a variable number of replicated updater processes. Various mutual exclusion protocols for accessing concurrent B-tree* structures are described in [1] and our example has been inspired by this work. This example exhibits many symmetric states mainly because all nodes inserted in the structure are identical. The example scales in the number of updater processes (N), B-tree order (K) and maximum depth of the structure (D). Table 2 shows the results.

Table 2: B-Tree* Example

N, K, D	States	Transitions	Speed (states/sec)	Memory (Mb)	Symmetry Reductions
3,2,2	28222	36689	9407	10.0	no
	5861	7789	5861	3.4	yes
2,2,3	171329	197864	10078	82.7	no
	6664	8091	6664	4.9	yes
2,4,3	2.09408e+06	2.41055e+06	408	> 1270	no
	18215	21830	6071	12.0	yes

The verification in the last case ($N = 2, K = 4, D = 3$) without symmetry reductions failed to terminate due to space limitations. Initially, the model checking process lasted 85 minutes before the virtual memory of the machine was exhausted. Applying symmetry reductions in this case made verification possible in only 10 seconds.

[1]The speed (states/second) reports are related to a Sun Ultra10, 300MHz machine with 1 Gb RAM, running SunOS 5.7.

6 Conclusions and Future Work

We have presented a state space reduction technique that exploits symmetries of heap configurations. The strategy is based on the idea that in common object-oriented languages, such as Java, different locations of objects are not observable to the program. Formally, this relates to bisimulation and strong preservation of properties. We described an algorithm that computes the representative of an equivalence class of symmetric states. The algorithm uses a sorting heuristic to complete in linear time. We proved that the strategy is canonic, achieving the best-case reduction. The technique described here has been implemented in a model checker for software. Although preliminary, our experience shows encouraging results regarding the role symmetry reductions can play in model checking programs written in high-level object-oriented languages, where symmetries arise due to the large number of orderings the objects can take within the heap.

Even though applying symmetry reductions to reduce the size of models has been intensively explored over the past decade, the new domain of software model checking seems to offer interesting perspectives for future research. One possible way of continuing this work is the use of partial order reductions [9] in combination with heap symmetries. The two techniques are related in that partial order reductions may detect symmetries that occur on execution traces, rather than states. Our intuition is that alias analysis could be incorporated in our model checker generator to improve the performance of already existing partial order reductions. This could reduce a-priori the number of symmetric states, i.e., before symmetry reductions are applied, improving the overall performance of the model checker.

References

[1] R. Bayer, M. Schkolnick. Concurrency of Operations on B-Trees, Acta Informatica, Vol.9. Springer-Verlag (1977) pp 1 – 21.

[2] E. M. Clarke, E. A. Emerson, A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Trans. on Programming Languages and Systems, 8 (1986). pp. 244 – 263.

[3] E. M. Clarke, T. Filkorne, S. Jha. Exploiting Symmetry In Temporal Logic Model Checking, Proc. 5th Conference on Computer Aided Verification (1993), LNCS 697, pp. 450 – 462.

[4] E. M. Clarke, E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic, LNCS 131 (1981), pp 52–71.

[5] J. Corbett, M. Dwyer, J. Hatcliff et al. Bandera: Extracting Finite-State Models from Java Source Code, Proc. 22nd International Conference on Software Engineering (2000).

[6] P. Cousot, R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, In Conference Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1997), pp 238–252.

[7] D. Dams, D. Bosnacki. A Heuristic for Symmetry Reductions with Scalarsets, Proc. FME 2001, LNCS 2021 (2001), pp 518–533

[8] E. A. Emerson, A. P. Sistla. Symmetry and Model Checking, Proc. 5th Conference on Computer Aided Verification (1993), LNCS 697, pp. 463 – 478.

[9] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems, Lecture Notes in Computer Science, Vol. 1032 (1996)

[10] P. Godefroid. Exploiting Symmetry when Model-Checking Software, Proc. FORTE/PSTV'99 (1999) pp 257 – 275

[11] M. S. Hecht, J. D. Ullman. A Simple Algorithm for Global Data Flow Analysis Programs, SIAM J. Computing 4 (1975), pp 519 – 532.

[12] G.J. Holzmann. Logic Verification of ANSI-C Code with Spin, Proc. 7th SPIN Workshop, LNCS 1885, pp. 131 – 147.

[13] G. Holzmann. The SPIN Model Checker, IEEE Trans. on Software Engineering, Vol. 23, No. 5 (1997), pp. 279-295.

[14] R. Iosif. Symmetric Model Checking for Object-Based Programs, Technical Report, KSU CIS TR 2001-5

[15] R. Iosif, R. Sisto. dSPIN: A Dynamic Extension of SPIN, Proc. 6th SPIN Workshop, LNCS 1680 (1999), pp 261 – 276

[16] R. Iosif, R. Sisto. Using Garbage Collection in Model Checking, Proc. 7th SPIN Workshop, LNCS 1885 (2000), pp 20 – 33

[17] C. N. Ip, D. L. Dill. Better Verification Through Symmetry, Formal Methods in System Design 9 (1996).

[18] D. Kozen. Results on the Propositional Mu-Calculus, Theoretical Computer Science 27 (1983), pp. 335–354.

[19] F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking, Proc. 8th SPIN Workshop, LNCS 2057 (2001), pp 80 – 102

[20] Z. Manna, A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, (1992).

[21] W. Visser, K. Havelund, G. Brat and S. Park. Model Checking Programs, Proc. 15th Conference on Automated Software Engineering (2000)