Symmetric Model Checking for Object-Based Programs

Radu Iosif

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 polynomial-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.

1 Introduction

Finite-state verification techniques, such as model checking [1], 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 [10] or LTL [9] 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 [14] and Java [16]. 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 more 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 [16] and software-oriented explicit-state model checkers [20, 17]. The former class of tools relies on applying various static analyses such as slicing [27] and abstract interpretation [11] in order to curb the state space explosion. Specific to the latter category are on-the-fly optimizations such as partial-order reductions [23] and symmetry reductions [2]. 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 [16] 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 identical data structures (objects) to reduce the complexity of the model checking algorithm. 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 [20] and JPF [17] 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 [5]. Structural symmetries induce an equivalence relation on states. In practice, determining the symmetry equivalence has been shown to produce a good under-approximation of the bisimulation relation. Unfortunately, the general problem of computing symmetry equivalence classes [2] (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 polynomial 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 is used to generate 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 [21, 18].

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 [20]. This is an extension of the well-known model checker SPIN [19], 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 10 Mbytes with a reduction in verification time from 30 minutes to 12 seconds.

1.1 Related Work

The idea of exploiting symmetries to reduce the state space in automatic verification is not new. Among the first to use symmetries in model checking were Clarke, Filkorn and Jha [2], Emerson and Sistla [3] and Ip and Dill [8]. These approaches mainly consider systems composed of a fixed number of active components (processors) [2], variables of a special symmetry-preserving data type (scalarset) [8] as well as structural symmetries of specification formulas [3]. Experimental results show reductions of memory requirements in model checking, 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 [24] applies symmetry reductions to stateless model checking of programs. Here symmetries are defined on sequences of transitions so that symmetric states can only be reached from the initial state by symmetric transition sequences. The combination between symmetry and partial order reductions has been considered in [24, 4].

Reducing the complexity of representatives computations has been tackled by the work of Dams, Bosnacki and Holenderski [12, 13]. 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 unbounded number of identical objects.

To our knowledge, the problem of exploiting heap symmetries in model checking has been informally addressed by Visser and Lerda in [18]. 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. 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 reduction framework that is to be used in the rest of the paper, Section 4 provides a formal model of computations with objects, Section 5 introduces the strong preservation correctness result, Section 6 describes the computation of the representative of a symmetry equivalence class, Section 7 describes the implementation of the toolset and provides experimental results, and last, Section 8 concludes the paper.

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 matches 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).

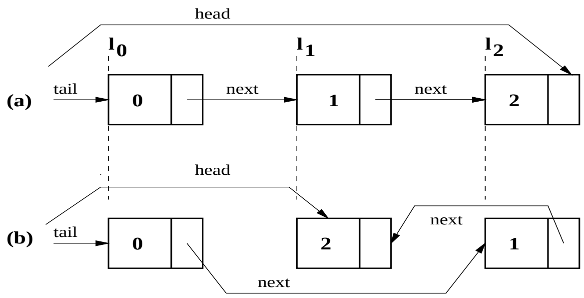

Figure 2: Message Queue Configurations

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. This situation will cause unwanted duplication of an entire region of the state space. A possible solution to this problem could be to switch the objects kept at locations l_1 and l_2 in Figure 2 (b) before storing the state into the state space. Switching object locations requires some references to be adjusted, in order to preserve program semantics. In our case, the `next` field of the first (0) object needs to be set to l_1 and similarly, the `next` field of the third (1) object needs to be set to l_2. This change is not observable to the program, since the roles played by the messages in some future computation are determined by their position in the queue and not by the relative memory positions. However, this will make the (b) configuration match the previously stored (a) configuration, since the heap states are now the same.

3 Background

In this section we present a slight variation of the symmetry group framework, first introduced in [2, 3]. The classical framework is based on the notion of group automorphism and deals 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). We let $I_n \in G_n$ denote the identity permutation, that is, the neutral group element.

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,
- $\rho : S \times \Sigma \to 2S$ 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 on 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 $s \equiv t$ both the following hold:*

- $\eta(s) = \eta(t)$,
- $\pi(s) = t$ *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. As a state in $M_{\equiv}$ represents possibly more than one state in 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 bisimilar states cannot be distinguished by formulas of Hennessy-Milner logic [5] and, consequently, of μ-calculus [7]. In our case, provided that M and $M_{\equiv}$ are bisimilar, a temporal logic model checker could interpret a temporal logic formula using $M_{\equiv}$ instead of M. This applies to μ-calculus formulas or any of its sub-logics, such as computation-tree logic (CTL) [10] or linear-time temporal logic (LTL) [9]. The barrier to apply this strategy in practice is related to the computation of the quotient transition system $M_{\equiv}$ from the original M. The general problem of finding the equivalence class of a state is known to be as hard as proving graph isomorphism, for which no polynomial-time solution is known to exist; this problem is also referred to in the literature as the *orbit problem* [2].

4 Object-based Programs

This section is concerned with the formal definition of a wide class of real-life software systems, based on the notion of objects. We introduce a simple language for describing object-based programs, that is, programs that manipulate a finite number of objects at a time by dynamically creating such objects, linking them into the reachable objects graph structure and modifying their internal state. The language is formally defined using abstract syntax and small-step operational semantics [6].

Figure 3 shows the abstract syntax of our object-based language (OBL). A program consists of a sequence of global variable declarations followed by a number of thread declarations. A variable can be of type boolean or reference. Reference variables represent handles to objects and can be used in access expressions or as a left hand side in new object creation statements. The structure of statements follows the guarded assignment model, where a guard is a boolean expression. We consider that all threads are created and started at the same moment, sometime in the initialization phase of the program and that each statement carries out an atomic computation. Even though each thread behaves deterministically, non-determinism may occur in each program due to different interleavings of its concurrent threads. For the sake of simplicity, there are no variables which are local to threads, and each action performed by some thread is an interaction between the thread and the global shared environment. Communication between threads follows the shared memory model. Furthermore in the definition of OBL we will not take into consideration type system related issues. We simply assume that every expression in the language can be correctly typed at compile-time.

An example of an OBL program is given in Figure 4. There are two threads and three variables in this program. The first thread, at program lines 1-3 creates a new object, makes `x` refer to it and sets its `f` field to `tt`. Next, the thread "waits" until there is an object assigned to the `y` variable in order to substitute the value of its `f` field with the logical negation of this value. The other thread, at program lines 4-6 behaves symmetrically, with variables `x` and `y` swapped.

The operational semantics of OBL is given in Figures 5 - 8. A program configuration s, h, n is a triple 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. 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

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
p & \in & \texttt{Programs} & x & \in & \texttt{Variables} \\
d & \in & \texttt{Declarations} & y & \in & \texttt{LeftHandSides} \\
b & \in & \texttt{Threads} & e & \in & \texttt{Expressions} \\
m & \in & \texttt{Statements} & a & \in & \texttt{Assignments} \\
t & \in & \texttt{Types} = \{\texttt{bool}, \texttt{ref}\} & c & \in & \texttt{Constants} = \{\texttt{tt}, \texttt{ff}, \texttt{nil}\}
\end{array}
$$

$$
\begin{array}{rcl}
p & ::= & d^{+}\ b^{+} \\
d & ::= & t\ x\ (:=c)?; \\
b & ::= & \texttt{begin}\ m^{+}\ \texttt{end} \\
m & ::= & e -> a; \\
y & ::= & x \mid y.x \\
e & ::= & c \mid y \mid !\ e \mid e = e' \mid (e) \\
a & ::= & y\ :=\ e \mid y\ :=\ \texttt{new}
\end{array}
$$

Figure 3: OBL abstract syntax

```
ref x := nil;
ref y := nil;
bool f := ff;

  begin
1 tt -> x := new;
2 tt -> x.f := tt;
3 !(y = nil) -> y.f := !y.f;
  end

  begin
4 tt -> y := new;
5 tt -> y.f := tt;
6 !(x = nil) -> x.f := !x.f;
  end
```

Figure 4: An OBL concurrent program

from their domains. The notation $A_\perp$ where A is a set, actually stands for $A \cup \{\perp\}$. We use structural operational semantics (SOS) rules to describe the meaning of the syntactic constructs from Figure 3. Rules $E1 \ldots E6$ define the meaning of expressions, rules $A1 \ldots A4$ refer to assignments, and rule $S1$ deals with statements. We make use of the overloaded $\vdash_t \Rightarrow$ operator as follows:

- $s, h, n\ \vdash_t \texttt{e} \Rightarrow v$ where $\texttt{e} \in \mathit{Expressions}$ means that expression e evaluates to v, under the program configuration s, h, n.
- $s, h, n\ \vdash_t \texttt{w} \Rightarrow s', h', n'$ where $\texttt{w} \in \mathit{Assignments} \cup \mathit{Statements}$ means that thread t applying the assignment (statement) w to the program configuration s, h, n yields a possibly different program configuration s', h', n'.

The $[\![.]\!]$ notation is used for the denotation of literals in the language. Its meaning is defined as follows:

- for an identifier x $[\![\texttt{x}]\!]$ is the program variable denoted by x,
- $[\![\texttt{null}]\!] = \perp$,
- $[\![\texttt{tt}]\!] = \mathit{true}$,
- $[\![\texttt{ff}]\!] = \mathit{false}$

For the sake of simplicity, none of the semantic rules deals with run-time errors such as uses of null or non-initialized references in field access expressions. We simply consider that OBL programs are written with care, and that they will always terminate in a consistent configuration, which is of course not the case in real programs, but it suffices for our demonstrative purposes.

$$\begin{array}{lcl} l & \in & Locations \\ b & \in & Booleans = \{true, false\} \\ v & \in & Values = Locations \cup Booleans \\ s & \in & Stores = Variables \longmapsto Values_{\perp} \\ h & \in & Heaps = Locations \longmapsto Stores_{\perp} \end{array}$$

Figure 5: Semantic domains

$$\frac{}{s,h,n \ \vdash_t \ \mathtt{c} \Rightarrow [\![\mathtt{c}]\!]}\ (E1) \qquad \frac{s,h,n \ \vdash_t \ \mathtt{e} \Rightarrow v}{s,h,n \ \vdash_t \ (\mathtt{e}) \Rightarrow v}\ (E4)$$

$$\frac{s,h,n \ \vdash_t \ \mathtt{e} \Rightarrow b}{s,h,n \ \vdash_t \ !\mathtt{e} \Rightarrow \neg b}\ (E2) \qquad \frac{}{s,h,n \ \vdash_t \ \mathtt{x} \Rightarrow s([\![\mathtt{x}]\!])}\ (E5)$$

$$\frac{s,h,n \ \vdash_t \ \mathtt{e} \Rightarrow v \quad s,h,n \ \vdash_t \ \mathtt{e}' \Rightarrow v'}{s,h,n \ \vdash_t \ \mathtt{e} = \mathtt{e}' \Rightarrow v = v'}\ (E3) \qquad \frac{s,h,n \ \vdash_t \ \mathtt{y} \Rightarrow l}{s,h,n \ \vdash_t \ \mathtt{y.x} \Rightarrow h(l, [\![\mathtt{x}]\!])}\ (E6)$$

Figure 6: Expressions

$$\frac{s,h,n \ \vdash_t \ \mathtt{e} \Rightarrow v}{s,h,n \ \vdash_t \ \mathtt{x} := \mathtt{e} \Rightarrow [[\![\mathtt{x}]\!] \rightarrow v]s,h,n}\ (A1)$$

$$\frac{s,h,n \ \vdash_t \ \mathtt{e} \Rightarrow v \quad s,h,n \ \vdash_t \ \mathtt{y} \Rightarrow l}{s,h,n \ \vdash_t \ \mathtt{y.x} := \mathtt{e} \Rightarrow s, [(l, [\![\mathtt{x}]\!]) \rightarrow v]h,n}\ (A2)$$

$$\frac{\exists l_{new} \in Locations \ . \ h(l_{new}) = \perp}{s,h,n \ \vdash_t \ \mathtt{x} := \mathtt{new} \Rightarrow [[\![\mathtt{x}]\!] \rightarrow l_{new}]s, [l_{new} \rightarrow \lambda x.\perp]h, n+1}\ (A3)$$

$$\frac{\exists l_{new} \in Locations \ . \ h(l_{new}) = \perp \quad s,h,n \ \vdash_t \ \mathtt{y} \Rightarrow l}{s,h,n \ \vdash_t \ \mathtt{y.x} := \mathtt{new} \Rightarrow s, [(l, [\![\mathtt{x}]\!]) \rightarrow l_{new}][l_{new} \rightarrow \lambda x.\perp]h, n+1}\ (A4)$$

Figure 7: Assignments

Let us now turn back to the sample program in Figure 4 and (informally) attempt to describe two of its possible computations. In what follows, let t_1 denote the thread declared at program lines 1-3 and t_2 denote the thread at lines 4-6. We first assume the following scheduling scenario (lines 1,2,4,5): t_1 first executes lines 1,2 so it creates a new object, makes `x` refer to it and then sets its `f` field to `tt`. Let l_0 be the location of this object in the heap. Then t_2 creates a second object at a new location, say l_1, makes `y` refer to it and then sets its `f` field to `tt`. The program configuration after this execution is shown in Figure 9 (a). Second, let us consider the scheduling scenario 4,5,1,2; let l_0' and l_1' be the locations at which the two objects are allocated this time. The execution will end in the configuration shown in Figure 9 (b). It is easy to see that in a future configuration obtained from (a) by executing line 3 (Figure 9 (c)), the object at location l_1 will have its `f` field set to false, while the same will happen with the object at location l_0' in Figure 9 (d). This kind of reasoning lead us to the conclusion that configurations (a) and (b) belong to the same equivalence class, generated by a permutation acting on the state space of the program in Figure 4. Locations l_0, l_0' and l_1, l_1' are not necessarily equal, but if we assume they are so[1] we can figure out that the permutation $(0,1),(1,0)$ describes a symmetry relation on configurations.

[1] This assumption is not unrealistic, since most allocators will always return the first available location from a countable set of memory locations.

$$\frac{s,h,n\ \vdash_t \texttt{e} \Rightarrow \texttt{tt} \quad s,h,n\ \vdash_t \texttt{a} \Rightarrow s',h',n'}{s,h,n\ \vdash_t\ \texttt{e}-> \texttt{a} \Rightarrow s',h',n'}\ (S1)$$

Figure 8: Statements

In other words, configuration (a) can be obtained from (b) by applying this permutation to the objects stored in memory. The same holds for all successor configurations of (a) and (b) such as (c) and (d) in Figure 9.

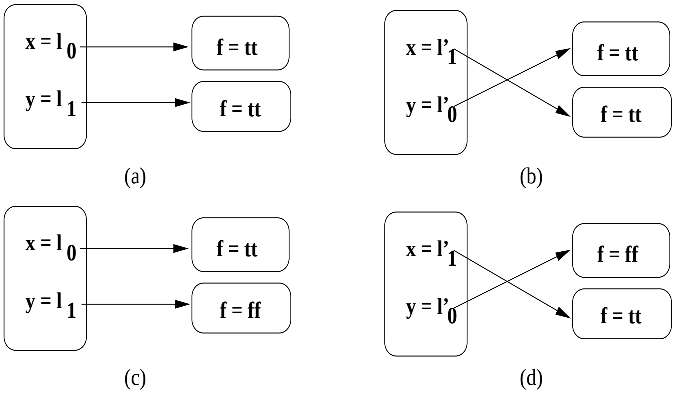

Figure 9: Symmetric Program Configurations

An important question to be answered is how to determine whether there exists a symmetry relation partitioning the state space of a program. We will restrict ourselves to heap symmetries, without ignoring that programs might exhibit many other symmetries that could be exploited in order to produce important state space reductions. Examples in the literature involve processes [12], variables of a special symmetry-preserving type (scalarset) [8] and symmetries of temporal logic specifications [3, 22, 25]. The reason we focus on heap symmetries when model checking object-based software is that, in dynamic programs, the number of objects manipulated is usually much larger than the number of other components, such as the threads. Consequently, the significantly larger number of permutations on objects increases the number of symmetrically equivalent states that occur during explicit-state model checking.

5 Symmetry Reductions

Having introduced a model of object-based computations in the previous section, we will now proceed with a formal definition of our symmetry relation, that we refer to as heap symmetry. The idea behind this is that different orderings of the same objects in the heap will always yield bisimilar states. In this section we formally define the meaning of applying a permutation on natural numbers to a program configuration (state) and prove that any such permutation induces a bisimulation. This enables us to apply the results in [7] to conclude that symmetry reductions strongly preserve temporal logic specifications i.e., for each formula f over a set of atomic propositions $\mathcal{P}$ and a symmetry relation $\equiv$, we have that $M \models f \Leftrightarrow M_{\equiv} \models f$.

First of all, let us translate the semantics introduced in Section 4 to the more general LTS model from Section 3, in order to be able to apply the framework recalled in Section 3 to the class of programs we have chosen to consider. Let $\mathcal{P} \subseteq \mathit{Expressions}$ that is, we allow only specification formulas that can be expressed using boolean expressions over program variables. The structure of expressions has been defined by the abstract syntax in Figure 3. Formally a program P will be represented by an LTS $M_P = \langle \Sigma_P, S_P, \rho_P, L_P, \eta_P \rangle$ where:

- $\Sigma_P = \mathit{Statements}$, the alphabet is the set of statements,
- $S_P = \mathit{Stores} \times \mathit{Heaps} \times \mathbb{N}$, each state s_P is a configuration s, h, n,

- $\rho_P(s_P, m) = \{s'_P \mid \exists t \in Threads\ .\ s_P\ \vdash_t m \Rightarrow s'_P\}$, a transition is fired by an arbitrary chosen thread executing a program statement,
- $L_P(s_P) = \{e \mid \forall t \in Threads\ s_P\ \vdash_t e \Rightarrow true\}$,
- $\eta_P(s, h, n) = n$.

In the remainder of this section we use the term state to refer also to program configurations.

In the preliminary section 3 we have considered that any permutation $\pi \in G_n$ on natural numbers induces a permutation on states. In order to give the permutation on states a formal expression, let the set $\mathcal{R} \subseteq Variables$ denote the set of variables declared of a reference type. For all $r \in \mathcal{R}$, such that $s, h, n\ \vdash_t r \Rightarrow v$, we have $v \in Locations$. Moreover, we assume that the set $Locations$ from Figure 5 is finite and countable that is, $Locations = \{l_0, l_1, \ldots l_{n-1}\}$ and, for all $l_i \in Locations$, let $\pi(l_i) = l_{\pi(i)}$ and $\pi(\perp) = \perp$. With these considerations, let $\pi(s_P) = \pi(s), \pi(h), n$, where $s_P = s, h, n$ and:

$$\pi(s) = \lambda r.\pi(s(r)),\ \forall r \in \mathcal{R} \tag{1}$$
$$\pi(s) = s,\ \forall r \notin \mathcal{R} \tag{2}$$
$$\pi(h) = \lambda l.\lambda r.\pi(h(\pi^{-1}(l), r)),\ \forall r \in \mathcal{R} \tag{3}$$
$$\pi(h) = \lambda l.h(\pi^{-1}(l)),\ \forall r \notin \mathcal{R} \tag{4}$$

Informally, the equations above say that, applying a permutation to a state, will permute all locations that represent 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 $\pi = (0,1)(1,0)$ to the variable `x` in the store from Figure 9 (a). Its value becomes $\pi(l_1) = l_0$ and the heap is changed by the permutation such that $\pi(h)(l_1) = h(\pi^{-1}(l_1)) = h(l_0)$. It is worth noticing that we choose to apply a permutation $\pi \in G_n$ only to those states whose number of components (objects) is n. In other words, π is well defined only on those states s such that $\eta(s) = n$.

We are now ready to prove that the symmetry relation induces a bisimulation on states. The result is obtained following several steps. First, we prove that the set of program states remains unchanged under state permutations. Second, that the denotation of boolean expressions remains the same if we apply permutations to states. Third, that assignments preserve the effect of permutations on states, and finally, the last theorem expresses the main result of this section. Let $S_n = \{s \in S_P \mid \eta_P(s) = n\}$.

Lemma 1 *For all $\pi \in G_n$ we have $\pi(S_n) = S_n$.*

Proof: Since $\eta(s) = \eta(\pi(s))$ for all $s \in S_n$. □

Lemma 2 *For all reference expressions $e \in Expressions$ and $s_P \in S_P$, $s_P\ \vdash_t e \Rightarrow l\ \wedge\ \pi(s_P)\ \vdash_t e \Rightarrow l'$ implies $l' = \pi(l)$, for all $\pi \in G_{\eta(s_P)}$.*

Proof: Let $s_P = s, h, n$. By induction on the structure of e, if e is:

1. `null` then, applying rule $E1$ to both conjuncts, and since $[\![\texttt{null}]\!] = \pi([\![\texttt{null}]\!]) = \perp$, we have $l' = \pi(l)$.
2. a store variable x then, by rule $E5$ applied to both conjuncts we have $l = s(x)$ and $l' = \pi(s)(x)$. Since x is a reference variable, by equation (1) we have $\pi(s)(x) = \pi(s(x))$, so $l' = \pi(l)$.
3. an access expression $y.x$, let $s_P\ \vdash_t y \Rightarrow k$, $\pi(s_P)\ \vdash_t y \Rightarrow k'$. By the induction hypothesis, we have $k' = \pi(k)$. Applying rule $E6$ to both conjuncts we get $l = h(k, x)$ and $l' = \pi(h)(k', x)$. Since x is a reference variable:
$$\begin{aligned} l' &= \pi(h)(k', x) \\ &= \pi(h(\pi^{-1}(\pi(k)), x)) \quad \text{by equation (3)} \\ &= \pi(h(k, x)) \\ &= \pi(l) \end{aligned}$$
4. `!e'` or `e' = e''`, since e cannot denote a reference expression, the implication holds trivially.
5. `(e')`, by the induction hypothesis and applying rule $E4$ to both conjuncts.

□

Lemma 3 *For all boolean expressions $e \in Expressions$ and $s_P \in S_P$, $s_P\ \vdash_t e \Rightarrow b\ \wedge\ \pi(s_P)\ \vdash e \Rightarrow b'$, implies $b = b'$, for all $\pi \in G_{\eta(s_P)}$.*

Proof: Let $s_P = s, h, n$. By induction on the structure of e, if e is:

1. a constant c then, by rule $E1$, we have $b = b' = [\![c]\!]$.
2. a store variable x then, by rule $E5$ we have $b = s(x)$. As e was supposed to be a boolean expression, x cannot be a reference variable and in this case $\pi(s)(x) = s(x)$. Applying rule $E5$ to the second conjunct leads to $b = b'$.
3. an access expression $y.x$, let $s_P \vdash_t y \Rightarrow l$, $\pi(s_P) \vdash_t y \Rightarrow l'$. By lemma 2 and since y denotes a reference expression, we have $l' = \pi(l)$. Applying rule $E6$ to both conjuncts we get $b = h(l, x)$ and $b' = \pi(h)(l', x)$. Since x is not a reference variable:

$$\begin{aligned} b' &= \pi(h)(l', x) \\ &= h(\pi^{-1}(l'), x) && \text{by equation (4)} \\ &= h(\pi^{-1}(\pi(l)), x) \\ &= h(l, x) \\ &= b \end{aligned}$$

4. !e', by the induction hypothesis and applying rule $E2$ to both conjuncts.
5. e' = e'' let $s_p \vdash_t e' \Rightarrow v$ and $s_p \vdash_t e'' \Rightarrow v'$. We need to consider three cases:
   (a) both e' and e'' denote reference expressions, then by lemma 2, $\pi(s_p) \vdash_t e' \Rightarrow \pi(v)$, $\pi(s_p) \vdash_t e'' \Rightarrow \pi(v')$ and $(v = v') = (\pi(v) = \pi(v'))$ since π is bijective.
   (b) none of e', e'' denote a reference expression, then $\pi(s_p) \vdash_t e' \Rightarrow v$ and $\pi(s_p) \vdash_t e'' \Rightarrow v'$.
   (c) e' is a reference expression and e'' is not a reference expression, then $v \neq v'$ holds for any $s_P \in S_n$, so it holds for $\pi(s_P)$, by lemma 1.
6. (e') by the induction hypothesis and applying rule $E4$ to both conjuncts.

□

Lemma 4 *For all $a \in Assignments$ and $s_P \in S_P$, we have $s_P \vdash_t a \Rightarrow s'_P \wedge \pi(s_P) \vdash_t a \Rightarrow s''_P$ implies $s'_P \equiv s''_P$, for all $\pi \in G_{\eta(s)}$.*

Proof: Let $s_P = s, h, n$. By induction on the structure of a, if a is:

1. x := e let $s_P \vdash_t e \Rightarrow v$, where $s_P = s, h, n$. We need to consider two cases:
   (a) both x and e are of type reference, by equation (1) we have $\pi([[\![x]\!] \to v]s) = [[\![x]\!] \to \pi(v)]\pi(s)$. By lemma 2, $\pi(s_P) \vdash_t e \Rightarrow \pi(v)$ and by rule $A1$ applied to the right conjunct, we obtain $s''_P = \pi(s'_P)$.
   (b) none of x and e is of type reference, by equation (2), we have $\pi([[\![x]\!] \to v]s) = [[\![x]\!] \to v]\pi(s)$. By rule $A1$ applied to the right conjunct, we obtain $s''_P = \pi(s'_P)$.
2. y.x := e let $s_P \vdash_t e \Rightarrow v$ and $s_P \vdash_t y \Rightarrow l$, where $s_P = s, h, n$. By lemma 2, and since y denotes a reference expression, $\pi(s_P) \vdash_t y \Rightarrow \pi(l)$. We need to consider two cases:
   (a) both x and e are of type reference:

   $$\begin{aligned} \pi([(l, [\![x]\!]) \to v]h) &= [(\pi^{-1}(\pi(l)), [\![x]\!]) \to \pi(v)]\pi(h) && \text{by equation (3)} \\ &= [(l, [\![x]\!]) \to \pi(v)]\pi(h) \end{aligned}$$

   By lemma 2, $\pi(s_P) \vdash_t e \Rightarrow \pi(v)$ and by rule $A2$ applied to the right conjunct, we obtain $s''_P = \pi(s'_P)$.
   (b) none of x and e is of type reference, by equation (4), $\pi([(l, [\![x]\!]) \to v]h) = [(l, [\![x]\!]) \to v]\pi(h)$. By rule $A2$ applied to the right conjunct, we obtain $s''_P = \pi(s'_P)$.
3. x := new by rule $A3$, if $s_P \in S_n$ then $s'_P \in S_{n+1}$. By lemma 1 we have $\pi(s_P) \in S_n$ and by rule $A3$ again $s''_P \in S_{n+1}$. Let $\pi' \in G_{n+1}$ be $[n \to n]\pi$. By (1) we have

   $$\begin{aligned} \pi'([[\![x]\!] \to l_{new}]s) &= [[\![x]\!] \to \pi'(l_{new})]\pi'(s) && \text{by equation (1)} \\ &= [[\![x]\!] \to l_{new}]\pi(s) && \text{by the definition of } \pi' \end{aligned}$$

   Also

   $$\begin{aligned} \pi'([l_{new} \to \lambda x.\perp]h) &= [\pi'^{-1}(l_{new}) \to \lambda x.\perp]\pi'(h) && \text{by equation (3)} \\ &= [l_{new} \to \lambda x.\perp]\pi(h) && \text{by the definition of } \pi' \end{aligned}$$

   By applying rule $A3$ to both conjuncts, we can conclude that $s''_P = \pi'(s'_P)$, and since $\pi' \in G_{n+1}$, $s''_P \equiv s'_P$.

4. `y.x := new` by rule $A4$, if $s_P \in S_n$ then $s'_P \in S_{n+1}$. By lemma 1 we have $\pi(s_P) \in S_n$ and by rule $A4$ again $s''_P \in S_{n+1}$. Let $\pi' \in G_{n+1}$ be $[n \to n]\pi$.

$$\begin{array}{ll} & \pi'([(l, [\![\mathtt{x}]\!]) \to l_{new}][l_{new} \to \lambda x.\,\bot]h, n+1) \\ = & [(\pi'^{-1}(l), [\![\mathtt{x}]\!]) \to \pi'(l_{new})][\pi'^{-1}(l_{new}) \to \lambda x.\,\bot]\pi'^{-1}(h) \quad \text{by equation (3)} \\ = & [(\pi^{-1}(l), [\![\mathtt{x}]\!]) \to l_{new}][l_{new} \to \lambda x.\,\bot]\pi(h) \quad \text{by the definition of } \pi' \end{array}$$

By applying rule $A4$ to both conjuncts, we can conclude that $s''_P = \pi'(s'_P)$, and since $\pi' \in G_{n+1}$, $s''_P \equiv s'_P$.

□

We can now proceed with proving that two symmetrically equivalent states are bisimilar. A direct consequence is that, given an LTS, the quotient of this LTS is related with it by a bisimulation, and since it is smaller, it can be used in model checking, instead of the original.

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

Proof: We need first to prove the following two points:

1. If $s \xrightarrow{m} s'$ then there exists $t' \in S$ such that $t \xrightarrow{m} t'$ and $s' \equiv t'$. If $s \equiv t$ then by definition 1, $\eta(s) = \eta(t) = n$ and there exists a permutation $\pi \in G_n$ such that $\pi(s) = t$. Let $s \xrightarrow{m} s'$ that is, $s \vdash_t m \Rightarrow s'$, where $m \in \mathit{Statements}$. Let m be `e -> a` with `e` denoting a boolean expression. The first precondition of rule $S1$ is $s \vdash_t e \Rightarrow \mathit{true}$ and, by lemma 3 we have $t = \pi(s) \vdash_t e \Rightarrow \mathit{true}$. The second precondition of rule $S1$ is $s \vdash_t a \Rightarrow s'$ and by lemma 4 we have t $= \pi(s) \vdash_t a \Rightarrow t'$ with $t' \equiv s'$. By rule $S1$ we have $t \xrightarrow{m} t'$ and $t' \equiv s'$.
2. $L(s) \subseteq L(t)$. Let $e \in L(s)$ be a boolean expression denoting an atomic proposition that holds in s. By lemma 3, e holds also in $t = \pi(s)$, so $e \in L(t)$.

Conversely, if $t = \pi(s)$, $\pi \in G_n$ and since G_n is a group, $s = \pi^{-1}(t)$ and $\pi^{-1} \in G_n$. The rest of the proof is identical, taking t instead of s and π^{-1} instead of π. □

6 Representative Computation

Since explicit-state model checkers for object-based programs [20, 17] 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 [13]. 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.

In general, orbit computations are expensive. A naive algorithm might attempt to exhaustively explore all possible permutations in the permutation group G_n in order to choose the representative of an orbit. Since $\|G_n\| = n!$ such an algorithm will usually not work in practice for a state in which the program heap keeps more than 10 objects. Examples of symmetry reductions [2, 3] usually sacrifice canonicity to reduce the complexity of orbit computations. The canonical heuristic described in [12] permutes the components (processes) of the state vector in order to find the lexicographically smallest state in the orbit. The strategy sorts the elements of the process array in order to find all the segments of identical components and then permutes all these segments independently. Since in our case, the number of identical objects (objects having the same data contents) can still exceed the bound (10) up to which we may want to explore all possible permutations, attempting to find the canonical representative by lexicographical sorting is not feasible.

Instead, our heuristic exploits the "shape" of the heap in order to find the canonical representative of a symmetry equivalence class. Let us represent a program configuration by 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 construct (and visit) the family of its depth-first spanning trees (DFST). A DFST [15] is a tree rooted at one of the entry nodes, trying to cover the longest paths in the graph possible. In fact, our choice for DFSTs is not a requirement, since any partitioning of the heap graph in a family of spanning trees will be able to give a canonical ordering of the heap objects, under a particular condition described below.

Let us consider again the OBL language described in Figures 5 - 8, $\mathcal{R} \in Variables$ be the set of all variables declared as of type reference and $\preceq_V \subseteq \mathcal{R} \times \mathcal{R}$ a total ordering of the reference variables declared in a program. The existence of such ordering is a key issue for the correctness of our strategy. The ordering can be found at compile-time, and one might consider for instance alphabetical order, declaration order, etc. This order entails a natural order on sequences of variables, denoted by $\preceq_V^* \subseteq \mathcal{R}^* \times \mathcal{R}^*$. 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 from $\mathcal{R}^*$ and x, y to denote variables from $\mathcal{R}$. We consider also a total order on the set *Locations*, denoted by in what follows by $\preceq_L$. As previously defined (Section 5), the behaviors of a program P are generated by an LTS $M_P = \langle \Sigma_P, S_P, \rho_P, L_P, \eta_P \rangle$ where a program state is a triple $s_P = s, h, n \in Stores \times Heap \times \mathbb{N}$. In the remainder of this section we make use of the following mappings.

$$\begin{array}{rcl} reach & : & S_P \times \mathcal{R}^* \to Locations_\perp \\ reach(s_P, v) & = & \begin{cases} s(x) & \text{if } v = x \\ h(reach(s_P, u), x) & \text{if } reach(s_P, u) \neq \perp \text{ and } v = u \circ x \\ \perp & \text{otherwise} \end{cases} \end{array}$$

and

$$\begin{array}{rcl} trace & : & S_P \times Locations \to \mathcal{R}_\perp^* \\ trace(s_P, l) & = & \begin{cases} min_V^* \{v \mid reach(s_P, v) = l\} & \text{if } reach(s_P, v) = l \text{ for some } v \in \mathcal{R}^* \\ \perp & \text{otherwise} \end{cases} \end{array}$$

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 in $\mathcal{R}^*$ might denote undefined locations (dangling chains), *reach* is allowed to return $\perp$ in order to express undefinedness. Conversely, $trace(s_P, l)$ returns the least sequence of reference variables that reaches l in state s_P or $\perp$ is no such sequence exists. The following lemma expresses the fact that *trace* is invariant under the state permutations previously defined by equations (1), (3) in Section 5.

Lemma 5 *For all $\pi \in G_{\eta(s_P)}$, $trace(s_P, l) = trace(\pi(s_P), \pi(l))$.*

Proof: We first prove that $\pi(reach(s_P, v)) = reach(\pi(s_P), v)$ by induction on the length of v. For the base step, let $v = x$:

$$\begin{array}{rcll} \pi(reach(s_P, x)) & = & \pi(s(x)) & \\ & = & \pi(s)(x) & \text{by (1)} \\ & = & reach(\pi(s_P), x) & \end{array}$$

For the inductive step let $v = u \circ x$ where $\pi(reach(s_P, u)) = reach(\pi(s_P), u)$:

$$\begin{array}{rcll} \pi(reach(s_P, u \circ x)) & = & \pi(h(reach(s_P, u), x)) & \\ & = & \pi(h(\pi^{-1}(\pi(reach(s_P, u))), x)) & \\ & = & \pi(h)(\pi(reach(s_P, u)), x) & \text{by (3)} \\ & = & \pi(h)(reach(\pi(s_P), u), x) & \text{by inductive hypothesis} \\ & = & reach(\pi(s_P), u \circ x) & \end{array}$$

The conclusion follows:

$$\begin{array}{rcl} trace(\pi(s_P), \pi(l)) & = & min_V^* \{v \mid reach(\pi(s_P), v) = \pi(l)\} \\ & = & min_V^* \{v \mid \pi(reach(s_P, v) = \pi(l)\} \\ & = & min_V^* \{v \mid reach(s_P, v) = l\} \\ & = & trace(s_P, l) \end{array}$$

□

Next, we consider that the sorting of reachability traces induces a permutation on states, defined as follows:

Definition 4 *A permutation $\pi_{sort} \in G_{\eta(s_P)}$ is called "sorting" for s_P iff for all $l_1, l_2 \in Locations$ such that $trace(s_P, l_1) \neq \perp$ and $trace(s_P, l_2) \neq \perp$, the following holds:*

$$\pi_{sort}(l_1) \preceq_L \pi_{sort}(l_2) \iff trace(s_P, l_1) \preceq_V^* trace(s_P, l_2)$$

The following lemma describes how composing permutations on states affects the sorting property.

Lemma 6 *Let $\pi_{sort}, \pi \in G_{\eta(s_P)}$ be two state permutations. π_{sort} is sorting for $\pi(s_P)$ iff $\pi_{sort} \circ \pi$ is sorting for s_P.*

Proof: For all $l_1, l_2 \in Locations$:

$$\begin{array}{lcll} (\pi_{sort} \circ \pi)(l_1) \preceq_L (\pi_{sort} \circ \pi)(l_2) & \iff & \pi_{sort}(\pi(l_1)) \preceq_L \pi_{sort}(\pi(l_2)) & \\ & \iff & trace(\pi(s_P), \pi(l_1)) \preceq_V^* trace(\pi(s_P), \pi(l_2)) & \pi_{sort} \text{ sorting for } \pi(s_P) \\ & \iff & trace(s_P, l_1) \preceq_V^* trace(s_P, l_2) & \text{by lemma 5} \end{array}$$

□

Next, we prove that sorting permutations are unique.

Lemma 7 *If $\pi_1, \pi_2 \in G_{\eta(s_P)}$ are sorting for s_P then $\pi_1 = \pi_2$.*

Proof: For all $l_1, l_2 \in Locations$:

$$\begin{array}{lcll} \pi_1(l_1) \preceq_L \pi_1(l_2) & \iff & trace(s_P, l_1) \preceq_V^* trace(s_P, l_2) & \text{by definition 4} \\ & \iff & \pi_2(l_1) \preceq_L \pi_2(l_2) & \text{by definition 4} \end{array}$$

Let $L = \{l_0, \ldots, l_{\eta(s_P)-1}\}$ be the set of all reachable locations of state s_P. Let $L_1 = l_{i_0}, l_{i_1}, \ldots, l_{i_{\eta(s_p)-1}}$ and $L_2 = l_{j_0}, l_{j_1}, \ldots, l_{j_{\eta(s_p)-1}}$ be two orderings of L such that for all $0 \le m, n, k \le \eta(s_P)$ it is the case that:

$$\begin{array}{lcl} l_{i_k} & = & \pi_1(l_k) \\ l_{j_k} & = & \pi_2(l_k) \\ i_m \le i_n & \iff & \pi_1(l_m) \preceq_L \pi_1(l_n) \\ j_m \le j_n & \iff & \pi_2(l_m) \preceq_L \pi_2(l_n) \end{array}$$

Then we have that $i_m \le i_n \iff j_m \le j_n$ for all $0 \le m, n \le \eta(s_P) - 1$, therefore $i_k = j_k$ which leads to $\pi_1(l_k) = \pi_2(l_k)$, for all $0 \le k \le \eta(s_P) - 1$. □

Let us now turn the discussion back to symmetric states and equivalence classes, as previously defined in Section 3. The discussion concerning sorting permutations was meant to prove that a sorting permutation gives the canonical representative of a symmetry equivalence class. The following theorem expresses the main result of this section:

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

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

Proof: By definition 1, if $s \equiv t$ then $\eta(s) = \eta(t) = n$ and there exists $\pi \in G_n$ such that $t = \pi(s)$. If π_t is sorting for t, then by lemma 6 $\pi_t \circ \pi$ is sorting for s. Since π_s is also sorting for s, by lemma 7, $\pi_s = \pi_t \circ \pi$ which, by equations (1 - 4), leads to $\pi_s(s) = \pi_t(\pi(s)) = \pi_t(t)$. □

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 (Figure 1 (b)), since $trace(l_0) \preceq_V^* trace(l_2) \preceq_V^* trace(l_1)$, the sorting permutation is $\pi_{sort} = \{(0,0), (1,2), (2,1)\}$. As one might expect, the result of applying π_{sort} to the heap in Figure 1 (b) matches the state in Figure 1 (a).

In practice, we are left with the task of finding the sorting permutation for a state. By theorem 2, applying a state a sorting permutation yields the canonical representative of its symmetry equivalence class, therefore the sorting permutation can be used as the representative function in the reduced state space search algorithm [13]. In the rest of this section we present an algorithm that generates the sorting permutation of a program state. We analyze this algorithm and show that it generates a sorting permutation. The algorithm in Figure 10 is a variant of the depth-first spanning tree construction algorithm from [15]. Let us assume that the function $ordered : Stores \to \mathcal{R}^*$ returns, for a given store, the $\preceq_V$-ordered sequence of reference variables that have defined values in that store. As a convention from now on, for any two locations l_i and l_j, let $l_i \preceq_L l_j$ iff $i \le j$. That is, locations are indexed with respect to $\preceq_L$.

The DFST procedure visits all reachable locations in the program state, assigning each location its corresponding depth-first order number (line 7). When a sequence of recursive calls to DFST visits a location l, the values held by the local variable x along the sequence of calls form a sequence of reference

```
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 := store(x)
4         if mark[l] = unvisited
5         begin
6                 mark[l] := visited
7                 π_sort := [l → l_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 k := 0
17 DFST(s)
```

Figure 10: The Sorting Permutation Algorithm

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_1 has been visited by DFST before l_2, it is the case that $v_1 \preceq_V^* v_2$, where $v_1 = trace(s_P, l_1)$ and $v_2 = trace(s_P, l_2)$. 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 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_1 before a recursive call (line 9) will eventually reach l_2. 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_1 (l_2). Following that k is continuously incremented (line 8) an from the way π_{sort} has been built (line 7) it is the case that $k_1 \leq k_2$, and $\pi_{sort}(l_1) \preceq_L \pi_{sort}(l_2)$ follows.

The worst-case complexity for the algorithm in Figure 10 running in state s is $O(N^2)$, 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., $E \leq N \times (N-1)$. The number of calls to DFST (lines 9, 17) 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 $O(N+E) = O(N^2)$ time in the worst case.

Even if the π_{sort} mapping constructed by DFST respects definition 4, in practice it might not be a permutation at all! Since DFST only visits reachable locations, it generates permutations only when all locations are reachable in a state, or in other words, when the state does not contain garbage objects. 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 [21]. The solution we use in practice consists in freeing all unmarked (garbage) locations and reindexing the remaining reachable locations. This makes π_{sort} become a permutation again, and since it satisfies the sorting property, by theorem 2, applying π_{sort} to a state produces the representative of its heap symmetry orbit.

7 Implementation and Experience

We have implemented our heap reduction strategy in the dSPIN model checker [20]. Similar extensions could be applied to other verification systems that offer explicit support for the representation of a program's heap. 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 only. The reason we do not report on model checking for temporal logic specifications, is that we performed these preliminary experiments in order to obtain a benchmark of the performances of our symmetry reduction strategy.

The dSPIN model checker is designed for verification of software, providing a number of novel features on top of standard SPIN's state space reduction algorithms, e.g., partial-order reduction and state compression. The input language of dSPIN is a dialect of the PROMELA language [19] offering, in addition, C-like constructs for allocating and referencing dynamic data structures. On-the-fly garbage collection is also supported [21]. Besides the implicit detection of deadlock and assertion violations, the model checker will report run-time errors such as uses of null or dangling pointers, that are common place in object-oriented programs. Embedding such capabilities directly into the model checker's core is 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 [21] to generate sorting permutations while visiting the reachable objects graph. Interestingly enough, there is no need to run the sorting permutation algorithm (Figure 10) more often than the garbage collector. The reason is that the set of statements that might generate heap-symmetric states (e.g., dynamic object creation, pointer assignment) is a subset of the ones that potentially produce garbage objects. A practical drawback of our strategy is that applying a permutation to the objects in a state must be followed by the adjustment of references. This results in a slowdown of the state permutation routine that is constant in the number of pointer variables declared in the program. In order to overcome this problem, our implementation of symmetry reductions uses static structures called *pointer maps* that are built at compile-time by the verifier generator tool. These maps encode the relative positions of pointers in the state structure, allowing for fast localization. Pointer maps are also useful to detect run-time errors caused by uses of dangling pointers.

The first test case models a simply-linked list ordered by node keys. The list can be updated by an arbitrary number of concurrent processes. There are two types of updater processes in our example: inserters that add given keys into the list for a finite number of times (N), and extractors that remove nodes with a given key from the list. Each node in the list has its own lock used by updater processes to synchronize. We considered the case in which one inserter acts in parallel with one remover. Symmetries arise in this case because of different interleavings between the inserter and the remover process, causing different allocation orderings of nodes with same keys. The example scales in the maximum length of the list, which in case of one inserter, matches the number of insert actions (N). For each value of the N parameter, we compared the sizes of the model checking problems with and without symmetry reductions. The results are shown in Table 1[2].

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

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

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 in the last case ($N = 10$) without symmetry reductions failed to terminate due to space limitations: it required more than 1.2 Gbytes of memory. Instead, the symmetry-reduced version of the system has been model checked in 370 Mbytes.

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 [26] and our example has been inspired by this work. Each node in the tree has an associated lock used to ensure exclusive access to its data. When performing insertion into a node, an updater process holds only the locks to its ancestors considered to be "unsafe" (i.e., the safety property is verified for all nodes that do not split due to an insertion). When a certain depth of the tree is reached, the updater processes stop their execution, to avoid an unbounded growth of the data structure. This example exhibits symmetries mainly because all nodes inserted in the structure are identical. In this cases our algorithm gives the best (factorial) reductions. The example scales in the number of updater processes (N), B-tree order (K) and maximum depth of the structure (D). Previous experience with this test case has been reported in [20, 21]. The size of the state space has been reduced using symmetries. 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, requiring 12 Mbytes of memory.

8 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 used a simple object-based language to prove that heap-symmetric states are observationally equivalent. The formalism introduced is used next to describe an algorithm that computes the representative of an equivalence class of symmetric states. The algorithm uses a sorting heuristic to complete in polynomial 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 programs. 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.

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 in combination with heap symmetries to speed up the model checker. The two techniques are related in that partial order reductions also 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 the number of calls to the symmetry equivalence representative function, improving the run-time performance of the model checker.

References

[1] 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.

[2] 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.

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

[4] E. A. Emerson, S. Jha, D. Peled. Combining Partial Order and Symmetry Reductions, Proc. TACAS, LNCS 1217 (1997) pp 19–34.

[5] M. Hennessy, R. Milner. Algebraic Laws for Nondeterminism and Concurrency, JACM 32, (1985), pp. 137–161.

[6] M. Hennessy. The semantics of programming languages : an elementary introduction using structural operational semantics, Wiley (1990).

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

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

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

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

[11] 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.

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

[13] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric Spin, Proc. 7th SPIN Workshop, LNCS 1885 (2000), pp 1–19

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

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

[16] 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).

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

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

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

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

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

[22] K. Ajami, S. Haddad, J.M. Ilie. Exploiting Symmetry in Linear Time Temporal Logic Model Checking: One Step Beyond, Proc. TACAS98, LNCS 1384 (1998) pp 52 – 67

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

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

[25] P. Sistla, P. Godefroid. Symmetry and Reduced Symmetry in Model Checking, Proc. 13th Conference on Computer Aided Verification (2001)

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

[27] F. Tip. A Survey of Program Slicing Techniques, Journal of Programming Languages, 3, (1995) pp 121 – 189