Symmetry reductions for model checking of concurrent dynamic software

Radu Iosif

▶ **To cite this version:**

Radu Iosif. Symmetry reductions for model checking of concurrent dynamic software. International Journal on Software Tools for Technology Transfer, 2004, 6 (4), pp.302-319. hal-01418878

HAL Id: hal-01418878
https://hal.science/hal-01418878

Submitted on 17 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Public Domain

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Symmetry Reductions for Model Checking of Concurrent Dynamic Software

Radu Iosif[1]

Verimag, Centre Equation,
2 Avenue de Vignate, 38610 Gieres, France
e-mail: Radu.Iosif@imag.fr

The date of receipt and acceptance will be inserted by the editor

Key words: software verification, symmetry reductions, orbit problem, temporal logic

1 Introduction

The increasing complexity in the design of concurrent software artifacts demands new validation techniques. Model checking [4] is a widespread technique for automated verification of concurrent systems that has been recently applied to the verification of software. Unfortunately, the use of model checking tools [13] is often limited by the size of the physical memory, due to the state explosion problem. In order to deal with this problem, various reduction techniques have been proposed in the literature. Among those, symmetry reductions [3], [8] and partial-order reductions [10], [25] have gained substantial credibility over the past decade. Both techniques are automatic and can be applied on-the-fly, during model checking. The reduction achieved can be significant, in the best cases exponential in the size of the state space.

Symmetry reductions exploit the structure of states in order to identify symmetric states that are generated by the model checker. The intuition behind these strategies is that the order in which state components (threads, objects) are stored in a state does not influence the observable behavior of the system. That is, the successors of two symmetric states are also symmetric. Many criteria have been proposed to decide whether two states are symmetric on-the-fly, without any information about the future states. They usually exploit the ordering of threads [6], communication channels and the structure of temporal logic formulas used to express correctness requirements [8]. A symmetry is an equivalence relation, and, ideally, the reduced state space will have only one state representing each symmetry equivalence class. Unfortunately, detecting all symmetries usually requires very expensive computations, that may make such reductions impractical, in general.

Partial order reductions exploit the commutativity of concurrent transitions, which lead to the same state when executed in different orders. The decision whether two transitions are independent, so that they can be safely swapped, is usually made using compile-time static analysis. In practice, this information is a conservative approximation of the real run-time independence. Using more static information about the system helps identifying more independent actions, however it is computationally more expensive. It has been shown [7] that symmetry and partial order reductions are orthogonal strategies and can be used in combination to achieve better verification results.

The main contribution of this paper is a framework for applying both reduction methods to a particular class of software, namely *dynamic concurrent* programs, for which the number of state components (objects, threads) is continuously modified as a result of their ongoing execution. This concept can be used to formalize the semantics of most high-level object-oriented programs, such as the ones written in Java or C++. We show how existing reduction techniques can be tailored to exploit the dynamic nature of software systems in order to achieve more effective verification results.

Preliminary results of this work have been presented in [19] and [20]. In [19] we present a canonical symmetry reduction that applies only to the heap of the program, while [20] is mostly concerned with the relation between heap and thread symmetry, as well as between symmetry and partial order reductions. We define a framework that allows us to express different symmetry reductions formally and compare their efficiency, in terms of canonical properties. Then we describe an explicit-state exploration algorithm that combines heap with thread symmetry reduction on-the-fly. Finally, we investigate further optimizations, by relating heap symmetries with

partial order reductions. Preservation of temporal logic properties is discussed throughout the paper. A prototype implementation of the ideas described in this paper has been done in dSPIN [17], an extension of SPIN [13], especially designed for software model checking. We performed a number of experiments with dSPIN, on two non-trivial test cases, in order to obtain a practical assessment of our ideas. More recently, a reduction method based on the theory presented in this paper has been implemented in the BOGOR software model checker [26]. A detailed description can be found in [27].

The rest of the paper is organized as follows. In the remainder of this section we present related work. Then, section 2 gives an informal introduction to the basic symmetry concepts by means of an example. Section 3 introduces the theoretical framework used to describe symmetry reductions. Section 4 describes two instances of this framework, namely heap and thread symmetries, using small-step operational semantics. Section 5 discusses the complexity of the Orbit Problem within the framework of dynamic systems. Section 6 presents algorithms for reduced state space search in presence of symmetries and discusses the combination of heap and thread symmetries on-the-fly. Section 7 relates symmetry with partial order reductions. Section 8 discusses implementation and presents experimental results, and Section 9 concludes.

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 [22]. These approaches consider systems composed of a fixed number of active components (memories, caches, processors) [3], variables of a special symmetry-preserving data type (scalarset) [22] as well as symmetries of temporal specifications [8]. The issue of sorting permutation to reduce the complexity of representatives computations has been addressed by the work of Dams, Bosnacki and Holenderski [6]. The problem of exploiting heap symmetries in software model checking has been informally addressed by Lerda and Visser in [24]. To our knowledge, they are the only other group that have addressed heap symmetries to date. Their approach looks attractive due to its simplicity, but no formal evidence of its canonical properties has yet been provided by the authors.

2 Motivating 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 concurrent language.

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

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

Fig. 1. Message Queue Example

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.

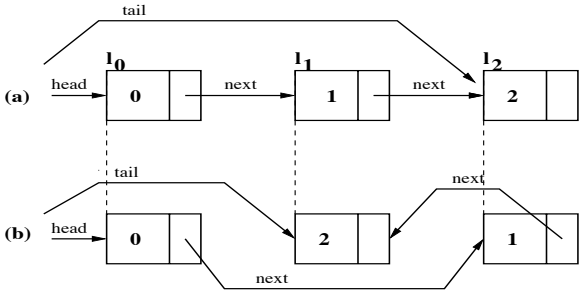

Fig. 2. Message Queue Configurations

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 next-free allocator, i.e., the first free memory location will always be returned by the allocator. A general abstraction of the underlying memory is given in terms of a discrete, totally 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 pointers are depicted as arrows labeled with field identifiers. 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 creating 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, a message with priority 1 will be created at this location. 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).

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. Notice however that the two states in Figure 2 can be obtained one from another by permuting the last two objects. Whenever this situation occurs, we say that the two states are symmetric.

3 Background

In this section we present some background notions regarding symmetry. In the classical literature [3], [8], symmetries are defined using the notion of *automorphism* i.e., internal isomorphisms that preserve the transition relation between states. However, using automorphisms to define symmetry fails to capture the *dynamic* aspect of computation. Indeed, when considering a program in which the number of state components (such as objects or threads) may experience an unbounded growth along an execution path, one cannot consider only one group of permutations as the group of system automorphisms. By doing so, one would fail to detect symmetries between successor states in which new components have been created. Instead, we consider a (possibly infinite) family of such groups and chose one at the time, by keeping track of the number of components in every state.

Formally, let us denote by G_n the set of all bijective functions (permutations) on the set $\{1, 2, \ldots, n\}$. It is easy to show that $(G_n, \circ, Id)$ is a group, where $\circ$ denotes functional composition and Id denotes the identity function. If we extend each permutation $\pi \in G_n$ to $\mathbb{N}$ i.e., considering that $\pi(i) = i$ for all $i > n$, we have the following subgroup relations:

$$G_i \subset G_j \iff i < j \tag{1}$$

Given an alphabet of *actions* Σ and a set of *atomic propositions* $\mathcal{P}$, we represent program executions by means of Kripke structures $K = (S, R, L)$, where:

- S is a set of states,
- $R \subseteq S \times \Sigma \times S$ is a transition relation, and $(s, \alpha, t) \in R$ is denoted by $s \xrightarrow{\alpha}_K t$,
- $L : S \to 2^{\mathcal{P}}$ is a function that labels states with sets of atomic propositions.

Unless otherwise specified, we will implicitly consider $K = (S, R, L)$ as the working structure throughout the paper. In the following developments, we will assume that states have some structure, namely, that each state is a finite set of *components* that can be partitioned in *types*. One can think for instance of active processes and heap-allocated objects as separate types of state components. We formally denote by $\mathcal{T}$ the set of all types and by $\mathcal{N}$ the family of functions $\eta_\tau : S \to \mathbb{N}$, one for each type $\tau \in \mathcal{T}$, such that, for each state $s \in S$, $\eta_\tau(s)$ is the number of τ-components in s. It is furthermore assumed that, during its execution, a program may only create new components and never delete the existing ones. Formally, if $s \xrightarrow{\alpha}_K t$ then $\eta_\tau(s) \leq \eta_\tau(t)$ for each $\tau \in \mathcal{T}$.

By analogy with natural numbers, we denote by G_S the set of all bijective functions $\pi : S \to S$. For any state s, and any permutation $\pi \in G_{\eta_\tau(s)}$, we denote by $\pi_\tau(s)$ the *application* of π only to the τ-components of s. Formally, for each type $\tau \in \mathcal{T}$ we consider an operator $()_\tau : G_n \to G_S$ that lifts a permutation on natural

numbers to a permutation on states. We assume that an $()_\tau$ operator meets the following requirements:

- π_τ preserves the number of state components i.e., $\eta_\omega \circ \pi_\tau = \eta_\omega$ for all $\tau, \omega \in \mathcal{T}$, and,
- different permutations on different types act independently i.e., $\pi_\tau \circ \pi'_\omega = \pi'_\omega \circ \pi_\tau$.

We are now ready to define the symmetry relation. By $\Pi_{i=1}^k \pi^i$ we denote the composition of permutations $\pi^1 \circ \pi^2 \circ \ldots \pi^k$.

Definition 1. Let $\mathcal{T}$ be the set $\{\tau_1, \tau_2, \ldots \tau_k\}$. We say that two states $s, t \in S$ are *symmetric*, denoted by $s \equiv t$, if and only if the following hold:

1. $L(s) = L(t)$,
2. there exists a sequence of permutations $\pi^i \in G_{\eta_{\tau_i}(s)}$, where $1 \leq i \leq k$, such that $\Pi_{i=1}^k \pi^i_{\tau_i}(s) = t$.

This definition is a slight generalization of the classical symmetry relation [3] [8], tailored to deal with multiple component types. The first item ensures that symmetry is equivalence with respect to the atomic propositions that can be observed in a state. Notice that, if two states s and t are symmetric, then $\eta_\tau(s) = \eta_\tau(t)$ for any $\tau \in \mathcal{T}$. Also, since composing different types of permutation applications is commutative, the order of types in $\mathcal{T}$ is not important. The following defines a restriction of symmetry to exactly one component type, by implicitly requiring that, for all types other than τ, the permutations be identities.

Definition 2. Let $\tau \in \mathcal{T}$ be a type. We say that two states $s, t \in S$ are *τ-symmetric*, denoted by $s \equiv_\tau t$, if and only if the following hold:

- $L(s) = L(t)$, and,
- there exists $\pi \in G_{\eta_\tau(s)}$ such that $\pi_\tau(s) = t$.

It is clear that $\equiv_\tau \subseteq \equiv$, for any component type $\tau \in \mathcal{T}$.

Using basic group theory, it can be shown that $\equiv$ is an equivalence relation on states[1] The equivalence class, also known as the *orbit*, of a state s is denoted by $[s]$. The *quotient* of a structure with respect to a symmetry relation is defined as follows:

Definition 3. Given a structure $K = (S, R, L)$ and a symmetry relation $\equiv \subseteq S \times S$, the *quotient* of K with respect to $\equiv$ is $K_{/\equiv} = (S', R', L')$, where:

- $S' = \{[s] \mid s \in S\}$,
- $R' = \{([s], \alpha, [t]) \mid (s, \alpha, t) \in R\}$,
- $L'([s]) = L(s)$, for all $s \in S$.

The states of a quotient structure are equivalence classes of states from the original structure and a transition occurs between two equivalence classes whenever a transition (labeled with the same action) occurs between states from the original structure. It is obvious, from the first point of Definition 1, that L' is well defined for the quotient structure. We remind that symmetric states have equal numbers of components, therefore by abuse of notation, we define $\eta_\tau([s]) = \eta_\tau(s)$, for each $\tau \in \mathcal{T}$ and each $s \in S$. Since the set S' is a (possibly non-trivial) partition of S, it is potentially more efficient to model check a temporal logic formula on $K_{/\equiv_\tau}$ instead of K, provided that they represent equivalent computations.

Let us turn back to the argument regarding the use of automorphisms in the definition of symmetry. Formally an automorphism of a structure K is a bijective function $\psi : S \to S$ such that, if $s \xrightarrow{\alpha}_K t$ then $\psi(s) \xrightarrow{\alpha}_K \psi(t)$, for any $s, t \in S$. Analogously, the set of all system automorphisms $(Aut_K, \circ, Id)$ forms a group. In the classical literature on symmetry [3], [8], state permutations are considered only if they can be shown to be system automorphisms. This condition can be however too restrictive to be applied to dynamic system. Assume, for simplicity, that $\mathcal{T} = \{\tau\}$ i.e., there exists only one type of symmetry in the system. Then we have $Aut_K \subseteq \bigcap_{s \in S} G_{\eta_\tau(s)}$. If we consider the existence of an initial state $s_0 \in S$ such that every other state $s \in S$ is reachable from s_0, we have $\eta_\tau(s) \geq \eta_\tau(s_0)$, and, by (1) we have $G_{\eta_\tau(s_0)} \subseteq G_{\eta_\tau(s)}$. Hence $Aut_K \subseteq G_{\eta_\tau(s_0)}$. Notice that, since most components are created as result of computation, it is usually the case that $G_{\eta_\tau(s_0)}$ is very small, therefore only a small number of system's symmetries can be identified.

To avoid this, in our approach we define equivalence of executions using directly the classical notion of *bisimulation* [12], strengthened with equivalence with respect to the set of atomic propositions $\mathcal{P}$:

Definition 4. Let $K_1 = (S_1, R_1, L_1)$ and $K_2 = (S_2, R_2, L_2)$ be Kripke structures over the set of actions Σ. A binary relation $\approx \subseteq S_1 \times S_2$ is a *bisimulation* if and only if, for all $s_1 \approx s_2$ and $\alpha \in \Sigma$, all the following hold:

- $L_1(s_1) = L_2(s_2)$,
- $\forall\, t_1 \in S_1 \,.\, (s_1, \alpha, t_1) \in R_1$ implies $\exists\, t_2 \in S_2 \,.\, (s_2, \alpha, t_2) \in R_2$ *and* $t_1 \approx t_2$,
- $\forall\, t_2 \in S_2 \,.\, (s_2, \alpha, t_2) \in R_2$ implies $\exists\, t_1 \in S_1 \,.\, (s_1, \alpha, t_1) \in R_1$ *and* $t_1 \approx t_2$.

If $\approx$ is total on S_1 and S_2 we say that K_1 and K_2 are bisimilar, and denote this by $K_1 \approx K_2$. To apply symmetry reductions to a structure K it suffices to prove that any symmetry $\equiv$ is a bisimulation. An important consequence is that, in this case, K and $K_{/\equiv}$ are bisimilar.

Lemma 1. *Given a structure $K = (S, R, L)$ and an equivalence relation $\approx \subseteq S \times S$ that is also a bisimulation. Then K and $K_{/\approx}$ are bisimilar.*

Proof: Directly from Definition 3 and Definition 4. □

[1] Reflexivity holds taking as π the identity, symmetry holds because for each $\pi \in G$, $\pi^{-1} \in G$ and associativity is a consequence of the fact that G is closed under functional composition.

It is known fact that bisimilar states cannot be distinguished by formulas of mu-calculus or any of its sublogics, such as computation-tree logic (CTL) or linear-time temporal logic (LTL) [4].

4 Semantic Aspects of Symmetry

This section is concerned with defining symmetry for a wide class of real-life software systems, namely *multi-threaded dynamic* programs. Basically, a multithreaded dynamic program can create new objects and spawn new threads along its execution. We will carry on future developments using a toy language that features both dynamic objects and multithreading. The first part of this section (4.1) will therefore define a Simple Pointer Language (SPL), that will be used in most proofs throughout the rest of the paper. Next (4.2), we define two types of symmetry, namely heap and thread symmetry for SPL.

4.1 A Simple Pointer Language

In the following we present an imperative multithreading language that captures the features of real-life object-oriented languages (C++, Java, etc.) related to the dynamic creation of objects and threads. Namely, we consider only three kind of actions: setting a pointer to null, assigning between two arbitrary pointers in the heap and creating a new object and assigning a pointer to it. Assignment actions are used in guarded statements, where guards consists of equalities of pointers or nullity tests. There are no types (in fact, all variables are of pointer type) and no variable declarations in our language. We consider that a variable is declared upon its first use.

Figure 3 shows the abstract syntax of SPL. There is a set of variable names $Vars$ and a set of thread names $ThreadNames$. A program P consists of a number of thread declarations. Each thread has a name T followed by a body composed of a sequence of control locations c. Each control location has attached one or more statements s. The first control location of a thread named T is, by convention, labeled $init_T$. A statement s is either a guarded assignment, a new object creation or a *start* action. Notice that the object and thread creation statements are implicitly assumed to have a true guard. A guard is a propositional logic expression f built out of observables p. An observable can either compare two pointer access paths σ and τ for equality or test a path for undefinedness.

We assume that there is always a designated `main` thread that starts running first. For the sake of simplicity we neglect the case selection and goto constructs. Also there are no variable declarations, assuming that a variable is defined the first time its name is used in the left-hand side of an assignment. As a convention, variable names that start with a capital letter denote global variables, while the other ones denote local variables. As one expects, threads evolve in parallel, communicating via global (shared) variables.

$$
\begin{aligned}
T, \text{main} &\in ThreadNames \\
c, init_T &\in Pc \\
u, v, U, V &\in Vars \\
\sigma, \tau &\in Vars^+
\end{aligned}
$$

$$
\begin{aligned}
P &:= (\text{thread } T \text{ begin } (c : s)^+ \text{ end})^+ \\
s &:= f \rightarrow \sigma = e \mid u = \text{new} \mid \text{start}(T) \\
e &:= \text{null} \mid \sigma \\
p &:= \sigma = \tau \mid \text{null}(\sigma) \\
f &:= p \mid f_1 \vee f_2 \mid \neg f_1 \mid \top
\end{aligned}
$$

Fig. 3. Abstract Syntax of SPL

Example Figure 4 (upper part) presents a sample SPL program that creates a list with two elements. The list is pointed to by the global variable H and linked with a selector n. Each node in the list is assigned a reference f to another cell. If each object is allocated at the next free memory location, this program always generates symmetric states, corresponding to the different interleavings of the two instances of thread T. Two such interleavings and the resulting states are shown in Figure 4 (lower part). By $T(0), T(1), \ldots$ we denote the actions corresponding to (only one of) the statements at location T in the first and second instances of thread T, respectivelly.

We can now sketch the semantics of SPL by describing the small-step operational semantics of its statements. The global semantics of the program will be the transition system obtained, as usual, by parallel composition of its active threads[2] and we shall not detail this construction here.

Figure 5 presents the semantic domains used to describe SPL. As usual, a store (a member of the *Stores* domain) is a partial mapping between variables and values. For simplicity reasons, it is assumed that variables can only take memory location values, from the set *Locs*. A heap (a member of the *Heaps* domain) is a pair whose first component is a partial mapping between locations and stores. We will refer to the stores in the range of this mapping as to *objects*. The second component of the heap is a location used to define the allocation policy; it keeps track of the *last* allocated memory cell. A thread (a member of the *Threads* domain) is a pair consisting of a program counter and a store for local variables. A program counter is the value of the current control location of the thread and a member of the *Pcs* domain.

[2] Since communication is by shared memory, we can assume that each thread's action alphabet is disjoint from the others.

```
thread main
begin
init_main: H = new
main_0:    start(T)
main_1:    start(T)
end

thread T
begin
init_T: v = new
T_0:    ¬ null(H) ∧ null(H.n) → H.n = v
        ¬ null(H.n) → H.n.n = v
T_1:    v.f = new
end
```

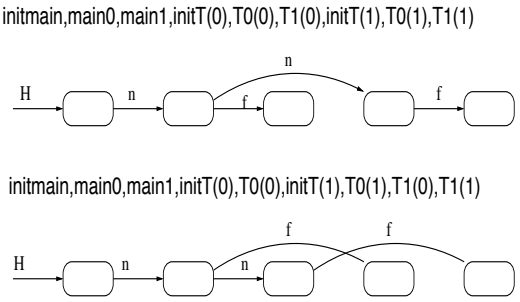

Fig. 4. SPL Program Example

In analogy with the heap, a pool (a member of the *Pools* domain) is a pair that keeps track of all active threads at a time; the first component of a pool is a partial mapping between thread identifiers (members of the *Tids* domain), while the second component holds the identifier of the last created thread. As a convention, for a partial mapping f we denote by $f(x) = \bot$ the fact that f is undefined in x. For a partially ordered set $(A, \preceq)$ we denote by $(A_\bot, \preceq)$ the partially ordered set $(A \cup \{\bot\}, \preceq')$, where $\bot \preceq' x$, for all $x \in A$, and $x \preceq' y$ if and only if $x \preceq y$, for all $x, y \in A$. We consider the *domain* of a partial function to be the set of points where it is defined i.e., $dom(f) = \{x \mid f(x) \neq \bot\}$. For a pair (a, b), let $(a, b)_{\downarrow 1} = a$ and $(a, b)_{\downarrow 2} = b$. We conclude the description of the semantic domains assuming the existence of the following *strict* and *total* orders:

- $\prec_v \subseteq Vars \times Vars$,
- $\prec_c \subseteq Pcs \times Pcs$ and a total function $next : Pcs \to Pcs$ that returns the next element with respect to $\prec_c$ i.e, the control label of the next statement to be executed; the set Pcs is supposed to be infinite and countable,
- $\prec_l \subseteq Locs \times Locs$ and a total function $new : Locs \to Locs$ that returns the next element with respect to $\prec_l$ i.e., the next free memory location; the set $Locs$ is supposed to be infinite and countable,
- $\prec_t \subseteq Tids_\bot \times Tids_\bot$ and a total function $start : Tids \to Tids$ that returns the next element with respect to $\prec_t$ i.e., the next free thread identifier; the set $Threads$ is supposed to be infinite and countable,

$$
\begin{aligned}
Stores &= Vars \rightharpoonup Locs \\
Heaps &= (Locs \rightharpoonup Stores) \times Locs \\
Threads &= Pcs \times Stores \\
Pools &= (Tids \rightharpoonup Threads) \times Tids
\end{aligned}
$$

Fig. 5. Semantic Domains

$$
\begin{aligned}
[\![\text{null}]\!]^x_{st} &= \bot \\
[\![\sigma]\!]^x_{st} &= \begin{cases} s(U) & \text{if } \sigma = U \\ (p(x)_{\downarrow 2})(u) & \text{if } \sigma = u \\ h([\![\tau]\!]^x_{st}, v) & \text{if } \sigma = \tau v \wedge [\![\tau]\!]^x_{st} \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
&\text{where } st = (s, (h, l), (p, t)) \\
[\![\sigma = \tau]\!]^x_{st} &= ([\![\sigma]\!]^x_{st} = [\![\tau]\!]^x_{st}) \\
[\![\text{null}(\sigma)]\!]^x_{st} &= ([\![\sigma]\!]^x_{st} = \bot)
\end{aligned}
$$

Fig. 6. Denotations of Expressions

With the above definitions and assumptions, we define now a program state st to be an element of the $States$ set, defined as follows:

$$st = (s, (h, l), (p, t)) \in States = Stores \times Heaps \times Pools$$

The first component of a state is the global store, the second is a heap holding all existing objects, and the third one is a thread pool keeping all active threads.

Figure 6 gives the denotation of the SPL constructs that can be evaluated without side effects. For a fragment of abstract syntax `ast`, the operator $[\![\text{ast}]\!]^x_{st} \in Loc \cup \{true, false\}$ returns the value of the `ast` expression as evaluated by the thread referenced by tid x in state st. The denotational definitions are compositional, following the structure of expressions. The straightforward definitions for the `f` non-terminal are omitted for brevity reasons.

The denotation of `null` is $\bot$, as `null` cannot represent a valid heap location. We distinguish between global and local variables, as globals are evaluated on the shared store, while locals are evaluated on the current thread's local store. The denotation of a sequence of variable names σ is the location found after the traversal of the heap, or $\bot$, if the path is dangling. Pointer aliasing and undefinedness tests are defined as usual.

Finally, the structural operational rules from Figure 7 define the semantics of the three statements in SPL. Given the abstract syntax tree of a statement `ast`, the operator $st \vdash_x \text{ast} \rightsquigarrow st'$ describes the transformation of a state st under the execution of statement `ast` by thread x. For a better understanding of the rules in Figure 7, we can highlight the following commonality: each rule is

applicable only if the value of the program counter c of the current thread x matches the location of the statement occurring in the postcondition. All rules reflect the implicit change of control within the current thread.

Intuitively, rules (2), (3) and (4) deal with three semantically different cases of assignment, based on whether the left-hand side of the assignment is a global variable, local variable or a heap access expression. In the first case, the global store s is updated, in the second case we update the local store of the thread referenced by x, and in the third case the object pointed to by τ in the heap h is updated accordingly.

Rules (5) and (6) define the meaning of the object allocation actions, distinguishing the case when the newly allocated object is pointed to by a global variable (5) from the case when its location is assigned to a local variable (6). Both rules conclude that the heap h must be updated by adding a new empty object $\lambda v.\bot$ to a fresh location k. The assignment of k to the left hand side is handled as in the previous.

The last rule (7) defines the thread creation action `start`. A new thread is created, whose program counter is initialized with the first control location of T, namely $init_T$, and whose local store is empty. Notice that, unlike objects, threads are not pointed to by variables. Even if this condition might seem to be a limitation, it captures the essential difference between heap-allocated objects and dynamically created threads: an active thread causes *observable* state changes even if it is not referenced by a program variable, while an object can contribute to a global state change only if it is still referenced by at least one variable.

As a last remark, all allocator actions exploit the orders on the *Locs* and *Tids* sets, respectively. Namely, the next available element, as returned by the *new* and *start* functions, are used for allocation of fresh components. In the remainder of this paper we shall denote such allocation strategies as *next-free*.

$$\frac{st = (s,(h,l),(p,t))\ p(x) = (c,s')\ c' = next(c)\quad p' = [x \to (c',s')]p\ \ [\![f]\!]^x_{st} = true\ \ [\![e]\!]^x_{st} = m}{st \vdash_x c : f \to U = e \rightsquigarrow ([U \to m]s,(h,l),(p',t))} \quad (2)$$

$$\frac{st = (s,(h,l),(p,t))\ p(x) = (c,s')\ c' = next(c)\quad [\![f]\!]^x_{st} = true \qquad [\![e]\!]^x_{st} = m \qquad s'' = [u \to m]s'}{st \vdash_x c : f \to u = e \rightsquigarrow (s,(h,l),([x \to (c',s'')]p,t))} \quad (3)$$

$$\frac{\begin{array}{l} st = (s,(h,l),(p,t))\ p(x) = (c,s') \qquad c' = next(c)\\ p' = [x \to (c',s')]p\ \ [\![f]\!]^x_{st} = true \qquad [\![e]\!]^x_{st} = m\\ [\![\tau]\!]^x_{st} = k \qquad o = [v \to m]h(k)\end{array}}{st \vdash_x c : f \to \tau.v = e \rightsquigarrow (s,([k \to o]h,l),(p',t))} \quad (4)$$

$$\frac{st = (s,(h,l),(p,t))\ p(x) = (c,s')\ c' = next(c)\quad p' = [x \to (c',s')]p\ \ k = new(l)\ \ h' = [k \to \lambda v.\bot]h}{st \vdash_x c : U = \text{new} \rightsquigarrow ([U \to k]s,(h',k),(p,t))} \quad (5)$$

$$\frac{\begin{array}{l} st = (s,(h,l),(p,t))\ p(x) = (c,s')\ c' = next(c)\\ p' = [x \to (c',s')]p\ \ k = new(l)\ \ h' = [k \to \lambda v.\bot]h\\ s'' = [u \to k]s'\end{array}}{st \vdash_x c : u = \text{new} \rightsquigarrow (s,(h',k),([x \to (c',s'')]p,t))} \quad (6)$$

$$\frac{st = (s,(h,l),(p,t))\ \ p(x) = (c,s')\ \ c' = next(c)\quad t' = start(t)\ \ p' = [x \to (c',s')][t' \to (init_T,\lambda v.\bot)]p}{st \vdash_x c : \text{start(T)} \rightsquigarrow (s,(h,l),(p',t'))} \quad (7)$$

Fig. 7. Operational Semantics

4.2 Heap and Thread Symmetry

Having defined a model language that is both concurrent and dynamic, we can proceed with defining state symmetries. The rather generic Definition 1 is specialized for SPL, by defining state permutations. More precisely, let $\mathcal{T} = \{heap, thread\}$ be the set of state component types. Since the set *Loc* of memory locations was supposed to be countable, let $Locs = \{l_0, l_1, \ldots\}$ and $\pi(l_k) = l_{\pi(k)}$. A similar definition is given to the permutation of thread identifiers *Tids*. A permutation $\pi : A \to A$ is implicitly extended to $A_\bot$ by setting $\pi(\bot) = \bot$ i.e., we consider strict permutations only. With these considerations we have, for a state $st = (s,(h,l),(p,t))$:

$$\pi_{heap}(st) = (\widehat{\pi}(s),(\widehat{\pi}(h),l),(\widehat{\pi}(p),t)) \quad (8)$$

$$\widehat{\pi}(s) = \lambda v.\pi(s(v)) \quad (9)$$

$$\widehat{\pi}(h) = \lambda l v.\pi(h(\pi^{-1}(l),v)) \quad (10)$$

$$\widehat{\pi}(p) = \lambda t.(p(t)_{\downarrow 1},\widehat{\pi}(p(t)_{\downarrow 2})) \quad (11)$$

$$\pi_{thread}(st) = (s,(h,l),(\widetilde{\pi}(p),t)) \quad (12)$$

$$\widetilde{\pi}(p) = \lambda t.p(\pi^{-1}(t)) \quad (13)$$

Equation (8) defines the semantics of a heap permutation. Intuitively, the values of all variables in the global store (9), heap (10) and local stores (11) are permuted. Although not formally defined, applying a permutation to the local store (second component) of a thread has the same form as (9).

The objects in the heap need to be permuted by the inverse permutation, in order to consistently reflect these changes, see (10). The following Lemma 2 ensures that same dereferencing sequences still point to the same objects in a permuted heap.

Since threads are not referenced by variables, permuting threads (12) has a simpler form (13).

The number of heap allocated objects in a state $st = (s,(h,l),(p,t))$ is $\eta_{heap}(st) = dom(h)$ and the number of active threads is $\eta_{thread}(st) = dom(p)$. It is obvious that $\eta_\tau \circ \pi_\omega = \eta_\tau$, for all $\tau, \omega \in \{heap, thread\}$. Moreover we have $\pi_{heap} \circ \rho_{thread} = \rho_{thread} \circ \pi_{heap}$, since $\widehat{\pi}(\widetilde{\rho}(p)) =$

$\widetilde{\rho}(\widehat{\pi}(p))$, as shown below:

$$\begin{aligned}\widehat{\pi}(\widetilde{\rho}(p)) &= \widehat{\pi}(\lambda t.p(\rho^{-1}(t)))\\ &= \lambda t.(p(\rho^{-1}(t))_{\downarrow 1}, \widehat{\pi}(p(\rho^{-1}(t))_{\downarrow 2})\\ &= \widetilde{\rho}(\lambda t.(p(t)_{\downarrow 1}, \widehat{\pi}(p(t)_{\downarrow 2}))\\ &= \widetilde{\rho}(\widehat{\pi}(p))\end{aligned}$$

Having a language with a complete operational semantics, together with formal definitions of state permutations, completes the definition of state symmetry for SPL. In order to apply Lemma 1 to the symmetry relation defined here, it remains to be shown that it is indeed a bisimulation.

Lemma 2. *Let $\sigma \in Vars^+$ be a sequence of variable names, $st \in State$ be a state and $\pi \in G_{\eta_{heap}(st)}, \rho \in G_{\eta_{thread}(st)}$ be two permutations. Then the following hold:*

1. $[\![\sigma]\!]^x_{\pi_{heap}(st)} = \pi([\![\sigma]\!]^x_{st})$,
2. $[\![\sigma]\!]^{\rho(x)}_{\rho_{thread}(st)} = [\![\sigma]\!]^x_{st}$.

Proof: Let $st = (s, (h, l), (p, t))$ throughout this proof.

1. By induction on the length of σ. The base case is $|\sigma| = 1$. Then either $\sigma = U$ is a global variable, and in this case $[\![\sigma]\!]^x_{\pi_{heap}(st)} = \pi(s(U)) = \pi([\![\sigma]\!]^x_{st})$. Otherwise $\sigma = u$ is a local variable and $[\![\sigma]\!]^x_{\pi_{heap}(st)} = \widehat{\pi}(p(x)_{\downarrow 2})(u) = \pi(p(x)_{\downarrow 2}(u)) = \pi([\![\sigma]\!]^x_{st})$. For the induction step we have $\sigma = \tau v$ and $[\![\sigma]\!]^x_{st} = h([\![\tau]\!]^x_{st}, v)$. By the induction hypothesis, $[\![\tau]\!]^x_{\pi_{heap}(st)} = \pi([\![\tau]\!]^x_{st})$, therefore $[\![\sigma]\!]^x_{\pi_{heap}(st)} = \pi(h(\pi^{-1}([\![\tau]\!]^x_{\pi_{heap}(st)}), v)) = \pi(h([\![\tau]\!]^x_{st}, v)) = \pi([\![\sigma]\!]^x_{st})$.
2. Also by induction on the length of σ. The only interesting case is $\sigma = u$. Then we have $[\![\sigma]\!]^{\rho(x)}_{\rho_{thread}(st)} = \widetilde{\rho}(p(\rho(x)))_{\downarrow 2}(u) = p(\rho^{-1}(\rho(x)))_{\downarrow 2}(u) = [\![\sigma]\!]^x_{st}$.

□

An immediate consequence of Lemma 2 is that the denotation of each observable expression, as defined by the abstract syntax in Figure 6, is preserved by state permutations:

$$[\![f]\!]^x_{st} = [\![f]\!]^x_{\pi_{heap}(st)} = [\![f]\!]^{\rho(x)}_{\rho_{thread}(st)} \tag{14}$$

An interesting result is the dual of (14): any two states st and st' such that, for all observables f we have $[\![f]\!]^x_{st} = [\![f]\!]^x_{st'}$, are symmetric. In other words, any *observational equivalence* is a symmetry. However, giving the proof here would be outside the scope of this paper. The interested reader is referred to [21] for a proof.

The next step in proving that symmetries of SPL are indeed bisimulations is to show that the results of any action taken in two symmetric states are symmetric.

Lemma 3. *Let $st \in State$ be a state and* s *be a statement, as defined by the abstract syntax in Figure 3. Then for any* s*-successor st' of st ($st \vdash_x s \rightsquigarrow st'$) and any two permutations $\pi \in G_{\eta_{heap}(st)}, \rho \in G_{\eta_{thread}(st)}$ there exist two permutations $\pi' \in G_{\eta_{heap}(st')}, \rho' \in G_{\eta_{thread}(st')}$ such that $\pi_{heap}(\rho_{thread}(st)) \vdash_{\rho(x)} s \rightsquigarrow \pi'_{heap}(\rho'_{thread}(st'))$. Moreover, the following hold:*

1. *if* s *is not defined by either rule (5) or (6), then $\pi' = \pi$.*
2. *if* s *is not defined by rule (7), then $\rho' = \rho$.*

Proof: Let $st = (s, (h, l), (p, t))$ throughout this proof. By induction on the structure of the inference tree for s. If st' was obtained by an application of either rule (2), (3) or (4), then $\eta_{heap}(st') = \eta_{heap}(st)$ and $\eta_{thread}(st') = \eta_{thread}(st)$. In this case take $\pi' = \pi$ and $\rho' = \rho$. Let $m = [\![e]\!]^x_{st}$ throughout the proof. By Lemma 2 we have $[\![e]\!]^{\rho(x)}_{\pi_{heap}(\rho_{thread}(st))} = \pi([\![e]\!]^x_{st}) = \pi(m)$. By (14) we have $[\![f]\!]^x_{st} = [\![f]\!]^{\rho(x)}_{\pi_{heap}(\rho_{thread}(st))}$, so whenever s is enabled in st, it is also enabled in $\pi_{heap}(\rho_{thread}(st))$ and viceversa. If, in particular st' was obtained by an application of:

- rule (2), we apply the same rule in state $\pi_{heap}(\rho_{thread}(st))$ for thread $\rho(x)$, and since $[U \to \pi(m)]\widehat{\pi}(s) = \widehat{\pi}([U \to m]s)$, we have the result.
- rule (3), let $p(x) = (c, s')$. Applying the same rule in state $\pi_{heap}(\rho_{thread}(st))$ for thread $\rho(x)$, we obtain the result after the following simplifications:

$$\begin{aligned}&[\rho(x) \to (c', [u \to \pi(m)]\widehat{\pi}(s'))]\widetilde{\rho}(\widehat{\pi}(p))\\ &= [\rho(x) \to (c', \widehat{\pi}([u \to m]s'))]\widetilde{\rho}(\widehat{\pi}(p))\\ &= \widetilde{\rho}([x \to (c', \widehat{\pi}([u \to m]s'))]\widehat{\pi}(p))\\ &= \widetilde{\rho}(\widehat{\pi}([x \to (c', [u \to m]s')]p))\\ &= \widehat{\pi}(\widetilde{\rho}([x \to (c', [u \to m]s')]p))\end{aligned}$$

- rule (4), let $k = [\![\tau]\!]^x_{st}$. By Lemma 2 we have that $[\![\tau]\!]^{\rho(x)}_{\pi_{heap}(\rho_{thread}(st))} = \pi([\![\tau]\!]^x_{st}) = \pi(k)$. Applying the same rule in state $\pi_{heap}(\rho_{thread}(st))$ for thread $\rho(x)$, we obtain the result after the following simplifications:

$$\begin{aligned}[v \to \pi(m)]\widehat{\pi}(h)(\pi(k)) &= [v \to \pi(m)]\widehat{\pi}(h(k))\\ &= \widehat{\pi}([v \to m]h(k))\end{aligned}$$

If st' was obtained by an application of either rule (5) or (6), then $k = new(l)$ is a fresh location i.e., $h(k) = \perp$. In this case we have $\eta_{heap}(st') = \eta_{heap}(st) + 1$ and $\eta_{thread}(st') = \eta_{thread}(st)$. Let us take $\pi' = [k \to k]\pi$ and $\rho' = \rho$. If, in particular st' was obtained by an application of:

- rule (5), we apply the same rule in state $\pi_{heap}(\rho_{thread}(st))$ for thread $\rho(x)$. The result is obtained due to the simplification $\widehat{\pi'}([U \to k]s) = [U \to k]\widehat{\pi}(s)$.
- rule (6), we apply the same rule in state $\pi_{heap}(\rho_{thread}(st))$ for thread $\rho(x)$. Let $p(x) = (c, s')$ and notice that $\widehat{\pi'}([u \to k]s') = [u \to k]\widehat{\pi}(s')$ and $\widehat{\pi'}(p) = \widehat{\pi}(p)$. The result is obtained as follows:

$$\begin{aligned}&[\rho(x) \to (c', [u \to k]\widehat{\pi}(s'))]\widetilde{\rho}(\widehat{\pi}(p))\\ &= [\rho(x) \to (c', \widehat{\pi}([u \to k]s'))]\widetilde{\rho}(\widehat{\pi'}(p))\\ &= \widehat{\pi'}(\widetilde{\rho}([x \to (c', [u \to m]s')]p))\end{aligned}$$

Finally, if st' was obtained by an application of rule (7), then $\eta_{heap}(st') = \eta_{heap}(st)$ and $\eta_{thread}(st') = \eta_{thread}(st)+1$, since $t' = start(t)$ is a fresh thread identifier. We take $\pi' = \pi$ and $\rho' = [t' \to t']\rho$. Let $p(x) = (c, s')$. Applying rule (7) in state $\pi_{heap}(\rho_{thread}(st))$ for thread $\rho(x)$, we obtain the result after the following simplifications:

$$\begin{aligned}
&[\rho(x) \to (c', \widehat{\pi}(s'))][t' \to (init_T, \lambda v.null)]\widetilde{\rho}(\widehat{\pi}(p)) \\
&= \widetilde{\rho'}([x \to (c', \widehat{\pi}(s'))][t' \to (init_T, \lambda v.null)]\widehat{\pi}(p)) \\
&= \widehat{\pi}(\widetilde{\rho'}([x \to (c', s')][t' \to (init_T, \lambda v.null)]p))
\end{aligned}$$

□

The global semantics of an SPL program is expressed by a Kripke structure $K_{spl} = (State, \to_{spl}, L_{spl})$ where $st \to^s_{spl} st'$ if and only if $st \vdash_x s \rightsquigarrow st'$ for some thread $x \in Tid$ and some statement $\mathtt{s}$, and $L_{spl}(st) = \{f \in \mathcal{P} \mid \exists x \in Tid \,.\, [\![f]\!]^x_{st} = true\}$ for a predefined set $\mathcal{P}$ of observables. We recall here that the largest bisimulation ($\approx$) has been introduced by Definition 4.

Theorem 1. *Given two states $st, st' \in State$, if $st \equiv st'$ then $st \approx st'$.*

Proof: By (14) and the definition of L_{spl}, we have that $st \equiv st'$ implies $L_{spl}(st) = L_{spl}(st')$. The second point of Definition 4 follows by Lemma 3 and the definition of $\to_{spl}$. □

As said before, this result is what enables us to use heap and thread symmetries in order to reduce the state space of an SPL program, leading to a more efficient verification of temporal logic properties.

5 The Orbit Problem

The key issue in order to make use of symmetry in model checking is establishing whether two states are in the same orbit. In general, the aim is at defining a function which takes two states as arguments and returns true if and only if the two states are in the same orbit. The computational complexity of such a function is discussed in [3], by relating the Orbit Problem to the Graph Isomorphism Problem: given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, is there a bijection $\psi : V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $(\psi(u), \psi(v)) \in E_2$?

Theorem 2 (Clarke et al.). *The Orbit Problem is as hard as the Graph Isomorphism Problem.*

Proof: See [3]. □

However, the latter problem is known to be in NP[3] for unlabeled graphs and can be shown to be in P for deterministic labeled graphs.

The results presented in this section relate these classical complexity issues [3] to the previous definitions, regarding the semantics of dynamic concurrent programs. In particular, we will show that, for object-manipulating single-threaded programs, the orbit problem is linear, whereas for multi-threaded programs, the problem is in NP (5.1). In addition to that, a general framework to develop heuristic solutions for the Orbit Problem is presented (5.2). The framework uses sorting techniques to compute *orbit representatives*. One can also say that an instance of the Orbit Problem is solved whenever the representative function is *canonical*. We can therefore assess the quality of our solution based on sorting if we can estimate the performance of the sorting techniques. Finally (5.3), we present a heuristic to solve the orbit problem for heap.

[3] But hasn't been shown to be complete for its class. A closely related problem, the Subgraph Isomorphism Problem is known to be NP-complete.

5.1 Complexity Issues

Given a state $st = (s, (h, l), (p, t)) \in State$ we shall define a unique labeled graph G_{st} such that two states st and st' are symmetric if and only if their graphs are isomorphic. Such constructions are classical in static analysis, being known under the name of *shape graphs* [23], [29]. Formally, we define $G_{st} = (V_{st}, E_{st}, \iota)$ where:

- $V_{st} = dom(h) \cup dom(p) \cup \{\iota\}$ is the set of vertices.
- $E_{st} \subseteq V_{st} \times (Vars \cup \{\epsilon\}) \times V_{st}$ is the set of edges, namely:

$$\begin{aligned}
E_{st} = &\{(\iota, U, l) \mid l \in Loc \wedge s(U) = l\} \\
&\cup \{(\iota, \epsilon, t) \mid t \in Tid\} \\
&\cup \{(t, u, l) \mid t \in Tid \wedge p(t)_{\downarrow 2}(u) = l\} \\
&\cup \{(l_1, u, l_2)\} \mid l_1, l_2 \in Loc \wedge h(l_1, u) = l_2\}
\end{aligned}$$

- ι is the *root* node of the graph.

There is one node for each location that is defined in the heap and one node for each active thread. In addition, we consider a unique root node, distinct from all other nodes. There is one edge for each global variable that is defined in the state, one for each active thread and one for each variable connecting two objects. Each edge representing a variable is labeled with that variable's name, except for the thread edges who are labeled with a special symbol ϵ. Notice that the graph is *rooted* i.e., there are no incoming edges to the root node. In the following developments, we assume that each location in the shape graph is reachable from the root node. The following lemma captures a key property of shape graphs:

Lemma 4. *Let $st, st' \in State$ be two states. Then st and st' are symmetric if and only if $G_{st} = (V, E, \iota)$ and $G_{st'} = (V', E', \iota')$ are isomorphic.*

Proof: Let $st = (s, (h, l), (p, t))$ and $st' = (s', (h', l), (p', t))$ throughout the proof. "⇒" Assume the existence of two permutations $\pi \in G_{\eta_{heap}}(st)$ and $\rho \in G_{\eta_{thread}}(st)$ such that $st' = \pi_{heap}(\rho_{thread}(st))$. Then we have $s' = \widehat{\pi}(s)$, $h' = \widehat{\pi}(h)$ and $p' = \widetilde{\rho}(\widehat{\pi}(p))$. Let us define $\psi : V \to V'$

as follows: $\psi(\iota) = \iota'$, $\psi(l) = \pi(l)$ for each $l \in V \cap Loc$, and $\psi(t) = \rho(t)$ for each $t \in V \cap Tid$. We prove that ψ is indeed an isomorphism between G_{st} and $G_{st'}$. Obviously ψ is a bijection. It remains to be shown that it preserves indeed graph structure:

1. $(\iota, U, l) \in E$ iff $s(U) = l$ iff $s'(U) = \pi(s(U)) = \pi(l) = \psi(l)$ iff $(\psi(\iota), U, \psi(l)) \in E'$.
2. $(\iota, \epsilon, t) \in E$ iff $t \in dom(p)$ iff $\rho(t) \in dom(p')$ iff $(\psi(\iota), \epsilon, \psi(t)) \in E'$.
3. $(t, u, l) \in E$ iff $p(t)_{\downarrow 2}(u) = l$ iff $p'(\rho(t))_{\downarrow 2}(u) = \pi(l)$ iff $(\rho(t), u, \pi(l)) \in E'$ iff $(\psi(t), u, \psi(l)) \in E'$.
4. $(l_1, u, l_2) \in E$ iff $h(l_1, u) = l_2$ iff $h'(\pi(l_1), u) = \pi(l_2)$ iff $(\psi(l_1), u, \psi(l_2)) \in E'$.

"$\Leftarrow$" Let $\psi : V \to V'$ be an isomorphism between G_{st} and $G_{st'}$. It is sufficient to show that $\psi(\iota) = \iota'$, $\psi(l) \in Loc$ for all $l \in V \cap Loc$ and $\psi(t) \in Tid$ for all $t \in V \cap Tid$. Then we have two permutations $\pi \in G_{\eta_{heap}(st)}$ and $\rho \in G_{\eta_{thread}(st)}$, which are the restrictions of ψ to Loc and Tid respectively. The conclusion then follows from the points (1) to (4) above. Assume that $\psi(\iota) \neq \iota'$. It follows that either $\psi(\iota) = l' \in V' \cap Loc$ or $\psi(\iota) = t' \in V' \cap Tid$. Since there is no incoming edge to ι there should be no incoming edge to $\psi(\iota)$. This contradicts both cases, since there is at least one incoming edge to each $l' \in V \cap Loc$ by the assumption that each location in a shape graph is reachable from the root node, and there is an edge (ι', ϵ, t') to each $t' \in V' \cap Tid$. So $\psi(\iota) = \iota'$. Assume that $\psi(l) \notin Loc$, for some $l \in V \cup Loc$. Then either $\psi(l) = \iota'$ or $\psi(l) = t'$ for some $t' \in V' \cap Tid$. In the first case there exists at least one incoming edge to l and there is no incoming edge to ι', which leads to a contradiction. In the second case there exists an edge (ι', ϵ, t') but there is no incoming ϵ-edge to l, so we have again reached a contradiction. Therefore $\psi(l) \in Loc$. The assumption $\psi(t) \notin V' \cap Tid$ is disproved in a similar way. □

We have reduced the problem of deciding whether two states are symmetric to deciding whether two shape graphs are isomorphic. Since shape graphs are non-deterministic in general, the problem is in NP. Notice however that the only sources of non-determinism for shape graphs are the ϵ-transitions between the root node and the identifiers of the active threads. However, in case of sequential programs, the generated shape graphs are deterministic, since there exists only one such ϵ-transition. The isomorphism problem in this case is in P. Moreover, the algorithm presented in the next section is linear in number of objects, for the sequential case. This is namely due to the fact that the branching degree (i.e., maximum number of outgoing edges) of a shape graph is bounded by a constant, as the number of variable identifiers used in a program is constant.

5.2 Sorting Permutations

Given the previous complexity results, we must turn our attention to heuristic techniques that give partial solutions. We shall also restrain to explicit-state model checking techniques that require the computation of orbit representatives.

Definition 5. Given a structure $K = (S, R, L)$ and an equivalence relation $\equiv$, a function $r : S \to S$ is said to be a *representative function* for $\equiv$ if and only if, for all $s \in S$, we have $s \equiv h(s)$.

Given a structure and a representative function, the state exploration algorithm will generate the image of the original structure through the representative function on-the-fly. Two states are identified as symmetric, when their representatives match. The effectiveness of the reduction is given by the following property of representatives functions:

Definition 6. Given a structure $K = (S, R, L)$ and an equivalence relation $\equiv$, a representative function $r : S \to S$ for $\equiv$ is said to be *canonical* if and only if, for all $s, s' \in S$ we have $s \equiv s' \iff r(s) = r(s')$.

The main idea behind the heuristics used in this paper is to obtain state representatives by a sorting algorithm. Since states, in our setting, are not necessarily linear, sorting is not necessarily applied only to vectors, but also to graph structures. In a first step, we can abstract from the actual structure of states introducing the notion of *sorting permutation*:

Definition 7. Let $K = (S, R, L)$ be a structure and $\xi : S \times \mathbb{N} \times \mathbb{N} \to \{true, false\}$ be a partial boolean mapping. Given a state $s \in S$ and type $\tau \in \mathcal{T}$, a permutation $\pi \in G_{\eta_\tau}$ is said to be sorting for s with respect to ξ if and only if, for all $0 \leq i, j < \eta_\tau(s)$, $\pi(i) < \pi(j) \iff \xi(s, i, j) = true$.

In the following, we refer to the ξ function as to the *sorting criterion*. The reason why ξ is allowed to be partial is a rather technical formality: we are not interested in the values $\xi(s, i, j)$ where i or j are greater than $\eta_\tau(s)$. However, not every boolean mapping ξ allows for the existence of sorting permutations. Take for instance a symmetric mapping $\xi(s, i, j) = \xi(s, j, i)$ for all $i, j \geq 0$. If a sorting permutation exists, then we have

$$\pi(i) < \pi(j) \iff \xi(s, i, j) \iff \xi(s, j, i) \iff \pi(j) < \pi(i)$$

which is a contradiction.

The intuition behind sorting criteria and sorting permutations are better explained by means of an example. Let $v : \{1, \ldots, n\} \to \mathbb{N}$ be a (finite) vector whose elements are natural numbers. Let $\xi(v, i, j) = v(i) < v(j)$. One can say that v' is a sorting of v with respect to ξ if, for all $i < j$ we have $\xi(v', i, j) = true$. The permutation (i, j) where $v(i) = v'(j)$ for all $1 \leq i, j \leq n$ is then the sorting permutation for ξ.

A sorting criterion is an abstract specification of a sorting problem. As said, we consider that an algorithm

solving the sorting problem will give, for a state, the representative of its orbit. Despite this level of abstraction, we can give necessary and sufficient conditions that need to be met by a sorting criterion in order for the induced representative function to be canonical. Since in practice it is easier to define a separate sorting criterion for each type of state component, we shall state the following result in particular for a given τ-symmetry (Definition 2). The generalization to full symmetry is immediate.

Theorem 3. *Let $K = (S, R, L)$ be a structure, $\tau \in \mathcal{T}$ be a type, $\equiv_\tau \subseteq S \times S$ be a τ-symmetry relation and $\xi : S \times \mathbb{N} \times \mathbb{N} \to \{true, false\}$ be a sorting criterion. Then the sorting permutations induced by ξ are canonical representative functions for $\equiv_\tau$ if and only if, for each state $s \in S$ and $0 \leq i, j < \eta_\tau(s)$, $i \neq j$, the following hold:*

- *ξ remains invariant under permutations of s, i.e, $\forall \pi \in G_{\eta_\tau(s)}$, $\xi(s, i, j) = \xi(\pi_\tau(s), \pi(i), \pi(j))$ and,*
- *ξ induces a strict total order on the set $\{1, \ldots, \eta_\tau(s)\}$ i.e., $[\xi(s, i, j) \vee \xi(s, j, i)] \wedge \neg[\xi(s, i, j) \wedge \xi(s, j, i)] = true$.*

Proof: $''\Leftarrow''$ Let s, t be two symmetric states in S. Then, by Definition 1, $\eta_\tau(s) = \eta_\tau(t) = n$ and there exists $\pi \in G_n$ such that $t = \pi_\tau(s)$. Let ρ and θ be the sorting permutations induced by ξ for s and t, respectively. We need to prove that $\rho_\tau(s) = \theta_\tau(t)$ or, equivalently, that $\sigma = \theta \circ \pi \circ \rho^{-1}$ is the identity permutation. Let us assume that σ is not the identity, therefore it can be expressed as a product of disjoint non-trivial cyclic permutations: $\sigma = (x_0, \ldots, x_k) \circ (y_0, \ldots, y_l) \circ \ldots$. Let us consider the first such cycle $(x_0, \ldots, x_k)$ that maps every x_i into $x_{(i+1) \bmod k}$. Without loss of generality, consider that $x_0 < \ldots < x_k$ i.e., we take the elements of the cycle in their natural order. Since ρ is sorting for s, we have that $\xi(s, \rho^{-1}(x_0), \rho^{-1}(x_k))$. By the first condition we have, equivalently, that $\xi(\pi_\tau(s), \pi(\rho^{-1}(x_0)), \pi(\rho^{-1}(x_k)))$ holds. Since $\sigma(x_k) = x_0$ and $\sigma(x_0) = x_1$, or equivalently, $\pi(\rho^{-1}(x_k)) = \theta^{-1}(x_0)$ and $\pi(\rho^{-1}(x_0)) = \theta^{-1}(x_1)$, we obtain $\xi(\pi_\tau(s), \theta^{-1}(x_1), \theta^{-1}(x_0))$ holds. Since θ was supposed to be sorting for $t = \pi_\tau(s)$ then $\xi(t, \theta^{-1}(x_0), \theta^{-1}(x_1))$ also holds, which contradicts the second condition of the theorem. $''\Rightarrow''$ Suppose ρ and θ are canonical representatives, i.e., $\rho = \theta \circ \pi$. Since ρ and θ are bijective, for all $i \neq j$, there exists $1 \leq x, y, z, w \leq n$, $x \neq y$ and $z \neq w$ such that $i = \rho(x) = \theta(z)$ and $j = \rho(y) = \theta(w)$. Since ρ is sorting for s and θ is sorting for $\pi_\tau(s)$ we have $\xi(s, x, y) = \xi(\pi_\tau(s), z, w)$ and, equivalently $\xi(s, x, y) = \xi(\pi_\tau(s), \pi(x), \pi(y))$ since $\rho = \theta \circ \pi$. This proves the first condition. For the second condition, observe that, for all x, y, $(\rho(x) < \rho(y)) \vee (\rho(y) < \rho(x))) \wedge (\neg(\rho(x) < \rho(y)) \vee \neg(\rho(y) < \rho(x)) = true$ therefore $(\xi(s, x, y) \vee \xi(s, y, x)) \wedge (\neg\xi(s, x, y) \vee \neg\xi(s, y, x)) = true$. This concludes our proof. $\square$

This result leverages the means for solving the difficult task of proving strategies canonical. It will be applied next, in order to compare two techniques, involving the detection of state symmetries induced by permutations of heap objects and threads. It will be also shown that the reduction strategy for heap symmetry is canonical, while the one for thread symmetry is not.

5.3 State Sorting

Considering the SPL semantics defined previously, we introduce two types of sorting problems: heap sorting and thread sorting. As said, in order to specify a sorting problem we need to define a sorting criterion $\xi : S \times \mathbb{N} \times \mathbb{N} \to \{true, false\}$. Then one can decide whether the reduction is canonical using Theorem 3.

To sort the heap, consider the set $Chains = Tids_\perp \times Vars^+$. A chain is a pair (t, σ) whose first element is a thread identifier or $\perp$ and second element is a sequence of variables. As a convention, $x = \perp$ if and only if the first symbol of σ must denote a global variable. We recall the existence of two strict and total orders: $\prec_v \subseteq Vars \times Vars$ and $\prec_t \subseteq Tids \times Tids$. Let $\prec^*$ be the pointwise order induced by $\prec_t$ and the lexicographical extension of $\prec_v$ to $Vars^+$ i.e., $(t, \sigma) \prec^* (t', \sigma')$ if and only if $t \prec_t t'$ and $\sigma \prec_v^* \sigma'$. The denotation of a chain (x, σ) in a state $st \in States$ is $[\![(x, \sigma)]\!]_{st} = [\![\sigma]\!]^x_{st}$ i.e., the location reached by thread t in st, following the path σ. Each location $l \in dom(h)$ can be therefore associated the set of all incoming chains. Moreover, these sets are disjoint, as one variable cannot point to two different locations. Since $\prec^*$ is total, each non-empty subset C of $Chains$ has a unique minimal element $\inf^* C \in C$. As a convention, $\inf^* \emptyset = \perp$. We can therefore associate each location in a state a unique chain as follows:

$$trace : State \times Locs \to Chains_\perp$$
$$trace(st, l) = \overset{*}{\inf}\{c \in Chains \mid [\![c]\!]_{st} = l\}$$

The sorting criterion for heap objects is denoted by ξ_{heap} and is defined as follows:

$$\xi_{heap}(\sigma, m, n) = trace(\sigma, l_m) \prec^* trace(\sigma, l_n) \qquad (15)$$

Since $\prec^*$ is strict and total, the second condition of Theorem 3 is met by ξ_{heap}. In order to show that ξ_{heap} induces a canonical representative function for $\equiv_{heap}$, it is sufficient to show that ξ_{heap} is invariant under state permutations i.e., $\xi_{heap}(st, m, n) = \xi_{heap}(\pi_{heap}(st), \pi(m), \pi(n))$. However, this is an easy consequence of the following lemma:

Lemma 5. *Given a state $st \in States$, for all $\pi \in G_{\eta_{heap}(st)}$ and $l \in Locs$, we have $trace(st, l) = trace(\pi_{heap}(st), \pi(l))$.*

Proof: By Lemma 2 we have $[\![c]\!]_{\pi_{heap}(st)} = \pi([\![c]\!]_{st})$ for any $c \in Chains$. Hence $trace(st, l) = trace(\pi_{heap}(st), \pi(l))$. $\square$

Consequently, the strategy that uses heap sorting is canonical, yielding optimal reductions. As an example, consider the situation in Figure 2, where the upper part depicts state st_a and the lower part depicts

state st_b. Considering that *head* and *tail* are global variables, and that the ordering on variable identifiers is: $head \prec_v next \prec_v tail$, we have:

$$trace(st_a, l_0) = (\bot, head)$$
$$trace(st_a, l_1) = (\bot, head, next)$$
$$trace(st_a, l_2) = (\bot, head, next, next)$$

Since $trace(st_a, l_0) \prec^* trace(st_a, l_1) \prec^* trace(st_a, l_2)$ the heap in st_a is already ordered. For st_b we have:

$$trace(st_b, l_0) = (\bot, head)$$
$$trace(st_b, l_1) = (\bot, head, next, next)$$
$$trace(st_b, l_2) = (\bot, head, next)$$

Since $trace(st_b, l_0) \prec^* trace(st_b, l_2) \prec^* trace(st_b, l_1)$, the sorting permutation for st_b is $\{(0,0),(1,2),(2,1)\}$.

The problem of sorting threads has however no easy solution. This fact is an unsurprising consequence of the previous complexity result. In fact, one can define a sorting criterion ξ_{thread} that meets the first point of Theorem 3 and induces a total order on $Tids$ which is not strict i.e., $\xi_{thread}(st, m, n) = \xi_{thread}(st, n, m)$ for some $m \neq n$. As an example, a heuristic proposed in [6], uses the values of the program counters in the sorting criterion. Let $c' \preceq_c c''$ stand for $c' \prec_c c'' \vee c' = c''$. For a state $st = (s, (h, l), (p, t))$, we define:

$$\xi_{thread}(st, m, n) = p(m) = (c', s') \wedge p(n) = (c'', s'') \wedge c' \preceq_c c'' \quad (16)$$

Two distinct threads having the same value of the program counter cannot be uniquely ordered by ξ_{thread}. Ideally, the number of such threads should be as small as possible. This can be achieved by strengthening the condition in the sorting criterion i.e., finding another criterion ξ'_{thread} such that $\xi'_{thread} \Rightarrow \xi_{thread}$. In practice, this is achieved by using more information about a thread than just the value of its program counter. This issue is further investigated in [27].

6 Model Checking with Symmetry Reductions

Given a representative function $r : S \to S$ i.e., a function that satisfies Definition 5, Figure 8 shows the basic depth first search state exploration algorithm with symmetry reductions, that generates on-the-fly the quotient structure of a program. The algorithm uses the primitives `add_state`, `add_transition` and `state_not_added` which are responsible of the bookkeeping of states and transitions. In particular, we consider that transitions are held in a set structure (implemented e.g. using a hash table with collision detection) in which case multiple additions of the same transition will be ignored.

The correctness of this algorithm is ensured by the fact that the representative of each state is symmetrical, and hence bisimilar, with the state itself. Therefore, when the representative state is already in the state

```
DFS(s)
add_state(r(s))
for each transition s --α--> t do
    add_transition(r(s) --α--> r(t))
    if state_not_added(r(t)) then DFS(t) fi
od
end DFS
```

Fig. 8. Symmetry Reduced Depth First Search

space, the algorithm can safely backtrack, since all successors of the representative state, which are bisimilar to the successors of the newly generated state, have been already explored.

In the remainder of this section, we will discuss effective computations of the r function. Based on the previous complexity results, r can be computed in polynomial time for single-threaded programs that allocate objects on the heap. One can attempt to simplify the problem, computing first a representative function r_{heap}, only for the heap of a state, disregarding the active threads. Second, another representative r_{thread} only for the active threads is computed and the final representative is the composition of the two functions. Next the two representative functions are applied to the current state, the result being a representative of the current state's orbit. Notice that, since the r_{thread} might not be canonical, it is possible to store more than one representative state per orbit. This affects the algorithm's efficiency but not correctness, for reasons discussed previously. Subsection 6.1 discusses the computation of canonical representatives for heap symmetry, while subsection 6.2 presents an algorithm that combines heap and thread symmetry, for more efficient reduction.

6.1 Computing Heap Sorting Permutations

Considering the heap sorting criterion defined by (15), the problem is to find a sorting permutation satisfying Definition 7. Since the ξ_{heap} criterion gives rise to canonical representative functions, there exists only one such permutation, according to Theorem 3. Proceeding under the simplifying assumption that each defined location in the heap is also reachable by a sequence of variables from some global or local variable, we show that the algorithm in Figure 9 computes indeed heap sorting permutations.

Intuitively, the algorithm in Figure 9 builds the depth-first spanning tree of a shape graph G_{st} in a state st, assigning each memory location $l \in Locs$ its corresponding depth-first order number k (line 5). A sorting permutation π_{heap} is build incrementally, in this way. We recall that a strict total order $\prec_v$ on the set of variables was assumed to exist.

The function $ordered : Store \to Vars^*$ returns, for a given store, the $\prec_v$-ordered sequence of variables that are defined in that store. This is in fact the key for the correctness of our algorithm: each time a store is visited,

```
Input: a state σ = (s,(h,l),(p,t))
Output: sorting permutation π ∈ G_{η_heap}(st)

SORT(store)
1 for next v from ordered(store) do
2     l_i = store(v)
3     if l_i not marked do
4        mark l_i
5        π = [i → k]π
6        k = k + 1
7        SORT(h(l_i))
8     od
9 od
end SORT

begin main
k = 0; π = λx. ⊥
10 SORT(s)
11 for each 0 ≤ t ≤ η_thread(st) do
12     (c,s') = p(t)
13     SORT(s')
14 od
end main
```

Fig. 9. Generation of Sorting Permutations for Heap Objects

the *next* variable in the $\prec_v$ order is chosen and its value (location) is explored. It is easy to see that, when a sequence of recursive calls to SORT visits a location l, the values held by the local variable v along the sequence of calls, prefixed with the thread identifier t chosen at line 11, forms the minimal chain c such that $[\![c]\!]_{st} = l$, or else, $c = trace(st, l)$. Our correctness claim is as follows:

Lemma 6. *A reachable location l is visited by the SORT algorithm before another reachable location l' in a state $st \in States$ if and only if $trace(st, l) \prec^* trace(st, l')$.*

Proof: "⇒" Assume that SORT reaches l before l'. Then SORT chooses to follow $trace(st, l)$ before $trace(st, l')$. Then either:

1. the SORT procedure that reaches l was started before the one that reaches l', either at line 10 or at line 13, in a previous iteration of the loop. In both cases we have $trace(st, l)_{\downarrow 1} \prec_t trace(st, l')_{\downarrow 1}$, and therefore $trace(st, l) \prec^* trace(st, l')$.
2. a call to SORT reaches l before a recursive call started at line 7 reaches l'. In this case $trace(st, l)$ is a prefix of $trace(st, l')$.
3. SORT chooses to follow $trace(st, l)$ before $trace(st, l')$ and the choice is made at line 1. Then $trace(st, l) = (t, \sigma)$, $trace(st, l') = (t, \sigma')$ and σ, σ' have a common prefix. Let x_1 (x_2) be the value chosen at line 1 in the first (second) case. Then $x_1 \prec_v x_2$ and therefore $trace(st, l) \prec^* trace(st, l')$.

"⇐" Let $trace(st, l) = (t, \sigma)$ and $trace(st, l') = (t', \sigma')$. Then $(t, \sigma) \prec^* (t', \sigma')$ either because:

1. $t \prec_t t'$, then the SORT procedure reaching l was started before the one reaching l' at either line 10 or 13, therefore l will be reached before l'.
2. $t = t'$ and σ is a prefix of σ' then l' is reached after l by a recursive call to SORT in line 7.
3. $t = t'$ and σ and σ' have a common prefix. Let τ be the longest such prefix and let l'' be a location such that $trace(st, l'') = (t, \tau)$. Also let x_1 be the first symbol on σ after τ and x_2 be the first symbol on σ' after τ. Since τ was the longest common prefix we have $x_1 \prec_v x_2$. Then the call SORT($h(l'')$) in line 7 will choose x_1 before x_2 and consequently reach l before l'. □

Since all locations were supposed to be reachable in st, the mapping built in line 5 is indeed a permutation. The fact that the permutation is indeed sorting for ξ_{heap} follows from Lemma 6 and the way the mapping is built (line 5), by pairing each location index with its corresponding depth-first number.

The algorithm in Figure 9 works properly, in principle, only in states where all locations are reachable i.e., in states that do not contain garbage objects. When a state contains garbage, the output of the algorithm might not be a permutation at all. This restriction can be however easily lifted in practice. Since the algorithm marks all reachable locations (line 4) in a state, it can be also used to perform on-the-fly garbage collection during model checking [18]. When the unmarked locations are freed, the remaining ones can be reindexed and the result will become a permutation.

Consider for example the configuration in Figure 2 (b) in which the indexes of all reachable locations occur increased by one and l_0 has become garbage. The SORT algorithm running in this state with the ordering $head \prec_v next \prec_v tail$ outputs the partial mapping $\pi = \{(1,0),(3,1),(2,2)\}$, which is undefined in 0. By eliminating the garbage location l_0, all indexes from the domain of π decrease by one, and the result is $\{(0,0),(2,1),(1,2)\}$, which is the needed sorting permutation.

The worst-case complexity for the algorithm in Figure 9 running in state st is $O(||G_{st}||)$, where $||G_{st}||$ is the size of the shape graph $G_{st} = (V, E, \iota)$ for st i.e, $||G_{st}|| = ||V|| + ||E||$. Notice however that $||E|| \leq ||Vars|| \cdot ||V||$, therefore the heap sorting algorithm requires at most $O(||G_{st}||) = O(||V|| + ||Vars|| \cdot ||V||) = O(||V||)$ time. If $st = (s, (h, l), (p, t))$ then $||V|| = ||dom(h)|| + ||dom(p)|| + 1$. In other words, computing heap sorting permutations can be done in time linear in the number of existing objects and threads.

6.2 Combining Heap and Thread Symmetry

Unlike heap-allocated objects, the set of active threads in a state has a linear structure, since we have assumed no references between threads. The problem of sorting threads reduces therefore to a classical vector sorting problem and we will not discuss it furthermore. Instead, we rather focus on finding an approximative solution to

the orbit problem, by generating sorting permutation independently, for the heap and threads, and combining the two permutations on-the-fly.

This idea originates with the observation that, given a state $st \in States$ and two permutations $\pi \in G_{\eta_{heap}(st)}$ and $\rho \in G_{\eta_{thread}(st)}$, their applications to st are commutative i.e. $\pi_{heap}(\rho_{thread}(st)) = \rho_{thread}(\pi_{heap}(st))$. However, using this straightforward composition to define the representative function h for the algorithm in Figure 8 faces the following problem: if π has been computed in σ using the sorting criterion ξ_{heap}, it might be the case that π is no longer sorting, according to ξ_{heap}, for $\rho_{thread}(st)$. As a result, applying the heap permutations computed according to ξ_{heap}, by the algorithm in Figure 9, does not give the canonical representatives for *heap*-symmetric states. The reason lies within the definition of ξ_{heap} (15), since a chain that reaches a location may be prefixed with a process identifier, and therefore the minimal chain for a location l, $trace(st, l)$, may depend on the order of processes. In other words, permuting processes may affect the canonical property of the heap symmetry reduction.

If SORT_{thread} denotes a sorting function for threads, usually implemented by a classical vector sorting algorithm, and SORT_{heap} is the function implemented by the algorithm in Figure 9, the representative function is computed as follows:

r(s)
begin
  $\rho = \text{SORT}_{thread}(s)$
  $\pi = \text{SORT}_{heap}(\rho_{thread}(s))$
  return $\pi_{heap}(\rho_{thread}(s))$
end

This function can be used with the algorithm in Figure 8 to generate the symmetry reduced structure of a program.

7 Symmetry versus Partial Order Reductions

Together with symmetries, partial order based methods are commonplace techniques for reducing the state space of a system. The basic idea is that, if we describe the execution of a system as a set of interleaving sequences of actions, one can define an equivalence relation on sequences and group them into equivalence classes. Each such equivalence class leads to a unique state in the system. For example, assume that actions α and β are *independent* i.e., the order in which they are executed is not important. Thus, the sequences $u\alpha\beta v$ and $u\beta\alpha v$ are equivalent. If a specification would not distinguish between equivalent sequences, it would be sufficient to consider only one representative out of each class, thus generating a much smaller state space. The problem is similar to the orbit problem, however the approaches differ in many ways. The contribution of this section is to further factor out the common points between the two reduction techniques, and present a combined reduction method that takes advantage of these similarities.

The previous work of Godefroid [11] also uses partial order information to detect symmetries between states, however it focuses mostly on flat programs, by defining permutations of actions and inferring that symmetric states are reached from the initial state by transition-symmetric paths.

A starting point is the observation that classical definitions of independence [10], [25], cannot consider allocator actions. The reason is that different interleavings of allocators lead to symmetric but different states. In practice, this corresponds to the very common situation in which various interleavings of threads that perform heap allocations generate heap symmetric states.

Our approach exploits the nature of dynamic programs that make use of the *next-free* allocation policy for which a semantics has been provided by the rules in Figure 7. The notion of independence is extended via symmetry to define *symmetric independence*. It can be shown that paths differing only by a permutation of adjacent symmetric independent actions lead to symmetric states. By conservatively exploiting this observation, when using partial order reductions in combination with symmetry reductions we can achieve better results if dynamic models of behavior are considered. The rest of this section is organized as follows: subsection 7.1 introduces the concepts of independence and symmetric independence, subsection 7.2 discusses the reduced state space exploration algorithms, and subsection 7.3 defines symmetric independence for the SPL language.

7.1 Symmetric Independence

For the rest of this section, let $K = (S, R, L)$ be a Kripke structure over a set of actions Σ. An action α is said to be *enabled* in state s if there exists a state t such that $s \xrightarrow{\alpha}_K t$. By $enabled_K(s)$ we denote the set of all actions enabled in s, according to the structure K. We can now introduce the concept of independent actions.

Definition 8. A symmetric irreflexive relation $I \in \Sigma \times \Sigma$ is said to be an independence relation for K iff for all $(\alpha, \beta) \in I$ and for each $s \in S$ such that $\alpha, \beta \in enabled_K(s)$, we have:

- if $s \xrightarrow{\alpha} t$ then $\beta \in enabled_K(t)$
- if for some $s', s'' \in S$, $s \xrightarrow{\alpha} s' \xrightarrow{\beta} t$ and $s \xrightarrow{\beta} s'' \xrightarrow{\alpha} t'$, then $t = t'$.

All partial order reduction algorithms [10], [25], [14] exploit (conservative under-approximations of) action independence. In practice, it has been shown that larger independence relations yield better partial order reductions. The contribution of this work to improving partial order reductions is based on defining and exploiting a weaker notion than the one from Definition 8.

Definition 9. Given a symmetry relation $\equiv$ on S, a symmetric irreflexive relation $I_S \in \Sigma \times \Sigma$ is said to be a symmetric independence relation for K iff for all $(\alpha, \beta) \in I_S$ and for each $s \in S$ such that $\alpha, \beta \in enabled_K(s)$, we have:

- if $s \xrightarrow{\alpha} t$ then $\beta \in enabled_K(t)$
- if for some $s', s'' \in S$, $s \xrightarrow{\alpha} s' \xrightarrow{\beta} t$ and $s \xrightarrow{\beta} s'' \xrightarrow{\alpha} t'$, then $t \equiv t'$.

The only change with respect to the Definition (8) is that, in I_S, two transitions are allowed to commute modulo symmetry. An independence relation is trivially a symmetric independence. Let us notice however that I_S can be much larger than I, since the number of states in a symmetry equivalence class can be exponential in the number of state components e.g., objects, processes. Dually, one can refer to the notion of *dependence*, which is defined as $D = \Sigma \times \Sigma \setminus I$. Similarly, we can define the notion of *symmetric dependence* as $D_S = \Sigma \times \Sigma \setminus I_S$. We can now formally relate the two notions of independence.

Lemma 7. *Given a symmetry relation $\equiv \subseteq S \times S$, I is a symmetric independence for K if and only if I is an independence for $K_{/\equiv}$.*

Proof: $"\Rightarrow"$ Let $(\alpha, \beta) \in I$ and $\alpha, \beta \in enabled_{K_{/\equiv}}([s])$. Let us prove the first point of Definition 8. Since, by Theorem 1, K and $K_{/\equiv}$ are bisimilar, $\alpha, \beta \in enabled_K(s)$. Let s' be the α-successor of s in K. Since I is a symmetric independence for K, $\beta \in enabled_K(s')$. Again since K and $K_{/\equiv}$ are bisimilar, $\beta \in enabled_{K_{/\equiv}}([s'])$. Since s' is the α-successor of s in K, $[s']$ is the α-successor of $[s]$ in $K_{/\equiv}$. For the second point of Definition (8), we have $s \xrightarrow{\alpha} s' \xrightarrow{\beta} t$ and $s \xrightarrow{\beta} s'' \xrightarrow{\alpha} t'$ in K. Since I is a symmetric independence for K, $t \equiv t'$, therefore $[t] = [t']$. $"\Leftarrow"$ This direction of the proof uses similar arguments. □

Let us notice that the possibility of using symmetric independence instead of classical independence is relative to the ability of building the quotient structure with respect to the symmetry relation considered. In other words, it is essential, for the symmetry considered in Definition 9, to compute a canonical representative function.

A second point of discussion concerns visibility of actions. An action α is said to be *invisible* with respect to a set of atomic propositions $P \subset \mathcal{P}$ iff, for all $s, t \in S$ such that $s \xrightarrow{\alpha} t$ it is the case that $L(s) \cap P = L(t) \cap P$. For a quotient structure $K_{/\equiv} = (S', R', L')$, as introduced by Definition 3, we have that $L(s) = L'([s])$ for each $s \in S$, therefore it is easily shown that an action is invisible in K if and only if it is invisible in $K_{/\equiv}$.

7.2 PO Reductions with Symmetric Independence

The main result of this section is based on a previous result by Emerson et al. [7]: performing partial order reduction on an already built quotient structure yields the same structure as using an algorithm that combines both partial order and symmetry reduction on-the-fly. Given a structure $K = (S, R, L)$ and a symmetry relation $\equiv \subseteq S \times S$, Figure 10 (a) shows the classical state space exploration algorithm with partial order reductions on the already built quotient structure $K_{/\equiv}$. This algorithm is however not meant to be used in practice. It is only presented here for the sake of future proofs.

```
DFS(s)
1 add_state(s)
2 push_state(s)
3 for each α in ample_a(s) do
4     for each transition s -α->_{K/≡} t do
5         add_transition(s -α-> t)
6         if state_not_added(t) then DFS(t) fi
7     od
8 od
9 pop_state()
end DFS
```

(a)

```
DFS(s)
1 add_state(s)
2 push_state(s)
3 for each α in ample_b(s) do
4     for each transition s -α->_{K/≡} t do
5         add_transition(r(s) -α-> r(t))
6         if state_not_added(r(t)) then DFS(t) fi
7     od
8 od
9 pop_state()
end DFS
```

(b)

Fig. 10. DFS with Partial Order and Symmetry Reductions

The algorithm keeps an explicit stack of states (accessed in lines 2 and 9). In order to generate the successor states (line 4), only transitions from a subset $ample_a(s) \subseteq enabled_{K_{/\equiv}}(s)$ are considered. The algorithm in Figure 10 (a) is correct[4] provided that the set $ample_a(s)$ meets the following constraints [4]:

- (C0-a) $ample_a(s) \neq \emptyset \iff enabled_{K_{/\equiv}}(s) \neq \emptyset$.
- (C1-a) on every path that starts with s in $K_{/\equiv}$, an action that is *dependent* on some action in $ample_a(s)$ cannot be taken before an action from $ample_a(s)$ is taken.
- (C2-a) if $ample_a(s) \subset enabled_{K_{/\equiv}}(s)$ then every $\alpha \in ample_a(s)$ is invisible.
- (C3-a) if $ample_a(s) \subset enabled_{K_{/\equiv}}(s)$ then, for every $\alpha \in ample_a(s)$ such that $s \xrightarrow{\alpha}_{K_{/\equiv}} t$, $t \notin Stack$.

Assume now that we are given a canonical representative for $\equiv$, say $r : S \to S$. Figure 10 (b) shows an algorithm that combines symmetry with partial order reductions on-the-fly. Notice that this algorithm works directly on the original structure K. A first difference with

[4] Property preservation for partial order reductions uses the notion of *stuttering path equivalence*, a weaker notion than bisimulation. For more details, the interested reader is referred to [25].

respect to the (a) algorithm is the use of the representative function r (lines 5 and 6) to generate the quotient structure on-the-fly. Second, the selection of transitions is made by a function $ample_b$ that uses the symmetric independence relation instead. We shall present first the conditions for $ample_b$ and then show their equivalence to the ones that define $ample_a$. This equivalence leads to the following fact: for each choice of transitions $ample_a(r(s))$ in $K_{/\equiv}$ it is always possible to choose the same transitions in K, by $ample_b(s)$, and viceversa.

In order to define the $ample_b$ function, we change conditions (C0-a) and (C2-a) into (C0-b), (C2-b) by syntactically replacing $ample_a$ with $ample_b$ and $K_{/\equiv}$ with K. The rules (C1-b) and (C3-b) are as follows:

- (C1-b) on every path that starts with s in K, an action that is *symmetric dependent* on some action in $ample_b(s)$ cannot be taken before an action from $ample_b(s)$ is taken.
- (C3-b) if $ample_b(s) \subset enabled_K(s)$ then for every $\alpha \in ample_b(s)$ such that $s \xrightarrow{\alpha}_K t$, then $r(t) \notin Stack$.

Since, by Lemma 1, K and $K_{/\equiv}$ are bisimilar, we have that $enabled_K(s) = enabled_{K_{/\equiv}}(r(s))$, which leads to conditions (C0-a) and (C0-b) being equivalent. Also, from the previous discussion concerning visibility of actions, we can conclude that (C2-a) and (C2-b) are equivalent. Since, by Lemma 7, a symmetric independence on K is an independence on $K_{/\equiv}$, we can infer that conditions (C1-a) and (C1-b) are equivalent. Equivalence of (C3-a) and (C3-b) can be shown as an invariant of the lockstep execution of the algorithms in Figure 10: at each step, the contents of the stack of algorithm (a) is the image of the contents of stack (b) via the representative function h.

Theorem 4. *Given a structure $K = (S, R, L)$ and a symmetry relation $\equiv \subseteq S \times S$, for each run of Algorithm (a) on the quotient structure $K_{/\equiv}$ there exists a run of Algorithm (b) on K such that the generated state spaces are the same, and viceversa.*

Proof: This proof is done between the lines of Theorem 19 from [7]. □

According to [25], partial order reduction preserves all formulas of the LTL_X (next-free LTL) logic. An algorithm for partial order reduction that preserves properties expressible in CTL^*_X can be found in [9]. As a consequence of this and Theorem 4, combining partial order based on symmetric independence with symmetry reductions, preserves all properties written as next-free temporal logic formulas.

The efficiency of the algorithm in Figure 10 (b) resides chiefly in the use of symmetric independence to choose from the set of enabled transitions. Since any symmetric independence is a non-trivial superset of the classical independence relation, our algorithm outperforms classical combinations of partial order and symmetry reductions [7].

7.3 Symmetric Independence for SPL

Notice that Definition 9, which introduces the largest symmetric independence with respect to a symmetry relation, is not very useful in practice. Indeed, given this definition, one cannot decide whether two actions are independent without checking if the successor states are symmetric, which in turn can be also expensive. We overcome this problem as usual, giving a conservative approximation of the symmetric independence relation for SPL.

In brief, we show that any two heap allocator actions, such that the right-hand sides are distinct variables, are symmetric independent with respect to the $\equiv_{heap}$ relation. We recall the small-step operational semantics of these actions, defined by rules (5) and (6) in Figure 7. The following fact is a direct consequence of the next-free allocation policy i.e., the first available memory cell is always allocated.

Lemma 8. *Let $K_{spl} = (State, \rightarrow_{spl}, L_{spl})$ be the transition system of an SPL program, $st \in States$ be a state and $\alpha = [a = new]$, $\beta = [b = new]$ be two statements, where a and b denote distinct variables. If $st \xrightarrow{\alpha}_{K_{spl}} st_1 \xrightarrow{\beta}_{K_{spl}} st_2$ and $st \xrightarrow{\beta}_{K_{spl}} st'_1 \xrightarrow{\alpha}_{K_{spl}} st'_2$, then $st_2 \equiv_{heap} st'_2$.*

Proof: The proof is a case analysis on the rules that define α and β. We shall give the proof only for the case in which both statements are applications of rule (5), the rest of the cases being similar. Let $Locs = \{l_1, l_2, \ldots\}$ be ordered according to $\prec_l$ and $st = (s, (h, l_k), (p, t))$, where l_k denotes the last allocated memory cell. Assuming that $new(l_i) = l_{i+1}$ for all $i \geq 1$, according to the next-free strategy, we have:

$$\begin{aligned}
st_1 &= ([a \rightarrow l_{k+1}]s, ([l_{k+1} \rightarrow o]h, l_{k+1}), (p, t)) \\
st_2 &= ([a \rightarrow l_{k+1}][b \rightarrow l_{k+2}]s, \\
&\qquad ([l_{k+1} \rightarrow o][l_{k+2} \rightarrow o]h, l_{k+2}), (p, t)) \\
st'_1 &= ([b \rightarrow l_{k+1}]s, ([l_{k+1} \rightarrow o]h, l_{k+1}), (p, t)) \\
st'_2 &= ([b \rightarrow l_{k+1}][a \rightarrow l_{k+2}]s, \\
&\qquad ([l_{k+1} \rightarrow o][l_{k+2} \rightarrow o]h, l_{k+2}), (p, t))
\end{aligned}$$

If we choose $\pi \in G_{k+2}$, $\pi = [k+1 \rightarrow k+2][k+2 \rightarrow k+1]Id$, it is easy to verify that $\pi_{heap}(st_2) = st'_2$, therefore $st_2 \equiv_{heap} st'_2$. □

Since allocator statements are always enabled in SPL i.e., we implicitly assume a true guard on those statements, the first point of Definition 8 is always met. Visibility of such actions depends on the set of program observables. We recall that an observable predicate in SPL can only compare two variables for equality or test for undefinedness. If a variable occurring on the right-hand side of an allocator statement does not occur in an observable term, we can safely conclude that the allocation is invisible. Hence, if $\mathcal{I}$ is the set of invisible allocator actions, then $\mathcal{I} \times \mathcal{I}$ is a symmetric independence for K_{spl}.

As a concluding remark, let us notice that the symmetric independence relation for SPL has been defined exclusively with respect to the heap symmetry. Since a canonical representative for heap symmetry can be effectively computed, the generation of the quotient structure with respect to $\equiv_{heap}$ is possible. This is exactly what enables us in this case to use Lemma 7 for proving the correctness of our approach.

8 Implementation and Experience

The heap symmetry and partial order reductions with symmetric independence have been implemented in the dSPIN model checker [17]. We performed experiments involving two test cases: the first one is a model of an ordered list shared between multiple updater threads, and 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 tests mainly to assess the effectiveness of our reduction techniques.

dSPIN is an automata theoretic explicit-state model checker designed for the verification of software. It provides 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 [18]. The presence of garbage collector algorithms in dSPIN made the implementation of heap symmetry reductions particularly easy. The algorithm used to compute sorting permutations is in fact an instrumented *mark and sweep* garbage collector. The explicit representation of states allowed the embedding of such capabilities directly into the model checker's core. This served to bridge 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 first test case represents a dynamic list ordered by node keys. The list is updated by two processes that use node locks to synchronize: an inserter that adds given keys into the list, and an extractor that removes nodes with given keys from the list. The example scales in the maximum length of the list (L).

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. The example scales in the number of updater processes (N), B-tree order (K) and maximum depth of the structure (D).

Table 1. Experimental Results

i. Ordered List Example

L	SI+SR	SR	SI	-
8	296	296	766	766
9	727	727	2.29e+03	2.29e+03
10	1.75e+03	1.75e+03	4.62e+03	4.62e+03

ii. B-Tree* Example

N, K, D	SI+SR	SR	SI	-
2, 2, 3	1.2	6.8	1.2	94
2, 4, 3	3	18	3	766
2, 4, 4	32	142	$\star$	$\star$

Symmetries arise in both examples because different interleavings of the updater processes cause different allocation orderings of nodes with the same data. The results of our experiments are shown in Table 1. The table shows the number of states (divided by 10^3 and rounded) generated by the model checker with standard partial order reduction only (-), with partial order based on symmetric independence only (SI), with symmetry reductions only (SR) and with combined partial order and symmetry reductions (SI+SR). We mark with $\star$ the cases when the model checker has exhausted both the physical and virtual memory of the computer.

In the first example (Ordered List) partial order reductions using symmetric independence do not contribute to the overall reduction of the state space. The reason is that the allocator statements in this model handle only global variables, being therefore labeled as "unsafe" by the dSPIN transition table constructor. On the contrary, in first two instances of the second example (Btree*) partial order reductions using symmetric independence manage to detect all heap symmetries arising as result of interleaving allocators, therefore symmetry reductions do not contribute any further to the overall reduction. The results show that combining partial order with symmetry reductions can outperform each reduction technique applied in isolation.

9 Conclusion

In this work, we have tackled issues related to the application of model checking techniques to software verification. Programs written in high-level programming languages have a more dynamic nature than hardware and network protocols. The size of a program state is no longer constant, as new components are added along executions. We have formalized this fact by means of semantic definitions of program states and actions. This semantics allows definition of various symmetry criteria for programs. We gave such criteria formal definitions, and described algorithms for on-the-fly symmetry reductions in automata theoretic model checking. In particular, we have discussed the combination of two orthogonal symmetry reductions, related to heap objects and processes. We have also shown how our heap symmetry re-

duction technique relates with partial order reductions. The emphasis is on how to adapt existing state space reduction techniques to software model checking. The ideas in this paper have been implemented in a software model checker that extends SPIN with dynamic features. Using this prototype implementation, a number of experiments have been performed. Preliminary results are encouraging, making us optimistic about the role symmetry and partial order reductions can play in enhancing model checking techniques for software.

References

1. R. Bayer and M. Schkolnick: Concurrency of Operations on B-Trees. Acta Informatica, Vol. 9 (1977) 1–21
2. D. Bosnacki: Enhancing State Space Reduction Techniques for Model Checking. PhD Thesis, Technical University of Eindhoven (2001)
3. E. M. Clarke, S. Jha, R. Enders and T. Filkorn: Exploiting Symmetry In Temporal Logic Model Checking. Formal Methods in System Design, Vol.9, No. 1/2 (1996) 77–104
4. E. M. Clarke, O. Grumberg and D. Peled: Model Checking. MIT Press (2001)
5. C. Courcoubetis, M. Y. Vardi, P. Wolper and M. Yannakakis: Memory-Efficient Algorithms for the Verification of Temporal Properties. Formal Methods in System Design, Vol. 1, No 2/3 (1992) 275–288
6. D. Dams, D. Bosnacki and L. Holenderski: A Heuristic for Symmetry Reductions with Scalarsets. Proc. Formal Methods Europe (2001) 518–533
7. E. Emerson, S. Jha and D. Peled: Combining Partial Order and Symmetry Reductions. Proc. Tools and Algorithms for Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1217 (1997) 19–34
8. E. Emerson and A. P. Sistla: Symmetry and Model Checking. Formal Methods in System Design, Vol.9, No. 1/2 (1996) 105–131
9. R. Gerth, R. Kuiper, D. Peled and W. Penczek: A Partial Order Approach to Branching Time Logic Model Checking. Proc. 3rd Israel Symposium on Theory on Computing and Systems (1995) 130–139
10. P. Godefroid: Partial-Order Methods for the Verification of Concurrent Systems. Lecture Notes in Computer Science Vol. 1032 (1996)
11. P. Godefroid: Exploiting Symmetry when Model-Checking Software. Proc. Formal Methods for Protocol Engineering and Distributed Systems (FORTE/PSTV) (1999) 257–275
12. M. Hennessy and R. Milner: Algebraic Laws for Nondeterminism and Concurrency. Journal of the ACM Vol. 32 (1985) 137–161
13. G.J. Holzmann: The SPIN Model Checker. IEEE Trans. on Software Engineering Vol. 23 (1997) 279–295
14. G. J. Holzmann and D. Peled: An Improvement in Formal Verification. Formal Description Techniques, Chapman & Hall, (1994) 197–211
15. G. Holzmann, D. Peled and M. Yannakakis: On Nested Depth First Search. Proc. 2nd SPIN Workshop (1996)
16. R. Iosif: Symmetric Model Checking for Object-Based Programs. Technical Report KSU CIS TR 2001-5 (2001)
17. R. Iosif and R. Sisto: dSPIN: A Dynamic Extension of SPIN. Proc. 6th SPIN Workshop, Lecture Notes in Computer Science Vol. 1680 (1999) 261–276
18. R. Iosif and R. Sisto: Using Garbage Collection in Model Checking. Proc. 7th SPIN Workshop, Lecture Notes in Computer Science Vol. 1885 (2000) 20–33
19. R. Iosif: Exploiting Heap Symmetries in Explicit-State Model Checking of Software. Proc. 16th IEEE Conference on Automated Software Engineering (2001) 254 – 261
20. R. Iosif: Symmetry Reduction Criteria for Software Model Checking. Proc. 9th SPIN Workshop, Lecture Notes in Computer Science Vol. 2318 (2002) 22 – 41
21. R. Iosif: An Observational Characterization of Heap Symmetry. Technical Report (2002) http://www-verimag.imag.fr/~iosif/completness.ps
22. C. Ip and D. Dill: Better Verification Through Symmetry. Formal Methods in System Design, Vol.9, No. 1/2 (1996) 41–75
23. N. D. Jones and S. S. Muchnick: Flow analysis and optimization of LISP-like structures. Program Flow Analysis: Theory and Applications, Chapter 4, 102–131. Prentice-Hall (1981)
24. F. Lerda and W. Visser: Addressing Dynamic Issues of Program Model Checking. Proc. 8th SPIN Workshop, Lecture Notes in Computer Science Vol. 2057 (2001) 80–102
25. D. Peled: All from One, One from All: on Model Checking using representatives. Proc. 5th Conference on Computer Aided Verification, Lecture Notes in Computer Science Vol. 697 (1993) 409–423
26. Robby, M. B. Dwyer, J. Hatcliff: Bogor: An Extensible and Highly-Modular Model Checking Framework. Proc. 4th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (2003)
27. Robby, M. B. Dwyer, J. Hatcliff, R. Iosif: Space-Reduction Strategies for Model Checking Dynamic Software, Electronic Notes in Theoretical Computer Science 89 No. 3 (2003)
28. Z. Manna, A. Pnueli: The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, (1992)
29. M. Sagiv, T. Reps, R. Wilhelm: Solving Shape-Analysis Problems in Languages with Destructive Updating, ACM Transactions on Programming Languages and Systems 20:1 (1998) 1–50