Automatic Discovery of Transition Symmetry in Multithreaded Programs Using Dynamic Analysis *

Yu Yang[1], Xiaofang Chen[1], Ganesh Gopalakrishnan[1], and Chao Wang[2]

[1] School of Computing, University of Utah, Salt Lake City, Utah, USA
[2] NEC Laboratories America, Princeton, New Jersey, USA

Abstract. While symmetry reduction has been established to be an important technique for reducing the search space in model checking, its application in concurrent software verification is still limited, due to the difficulty of specifying symmetry in realistic software. We propose an algorithm for automatically discovering and applying transition symmetry in multithreaded programs during dynamic model checking. Our main idea is using dynamic program analysis to identify a permutation of variables and labels of the program that entails syntactic equivalence among the *residual code of threads* and to check whether the local states of threads are equivalent under the permutation. The new transition symmetry discovery algorithm can bring substantial state space savings during dynamic verification of concurrent programs. We have implemented the new algorithm in the dynamic model checker `Inspect`. Our preliminary experiments show that this algorithm can successfully discover transition symmetries that are hard or otherwise cumbersome to identify manually, and can significantly reduce the model checking time while using `Inspect` to examine realistic multithreaded applications.

1 Introduction

Dynamic model checking [1,2,3] methods have proven promising for revealing errors in the implementation of real-world concurrent programs. They work on real applications and libraries, and side-step the high complexity of model construction and state capture by concretely executing the programs, and replaying the executions for covering the different thread interleavings. While several techniques have been proposed for reducing the complexity of dynamic model checking [4,5,6], one important technique – namely *symmetry reduction* – has yet to be fully explored during dynamic model checking.

In contexts where models of the systems are verified, symmetry reduction has already been shown highly beneficial for reducing the search space [7,8,9]. The basic approach taken in these works is finding and exploring the state space of a *quotient* transition system defined by an underlying equivalence relation – usually over the system states. This approach does not work well during dynamic model checking because the "state" consists of the runtime state of the threads being subject to concrete execution: capturing and canonicalizing such states is practically difficult or impossible. In dynamic model checking of concurrent programs, transition symmetry [10] has been used as an effective way of pruning of the search space; however, it requires the user to provide a permutation function in order to check whether two transitions are symmetric. In this paper,

* Supported in part by NSF award CNS-0509379, CCF-0811429, and Microsoft.

C.S. Păsăreanu (Ed.): SPIN 2009, LNCS 5578, pp. 279–295, 2009.

we present an algorithm to automate transition symmetry discovery by employing dynamic analysis. To our best knowledge, this is the first approach to automate transition symmetry discovery for the dynamic verification of realistic concurrent programs.

We first note that purely static analysis based approaches are often insufficient for detecting symmetries across threads. With static analysis, one may be tempted to conclude symmetry at the starting points of the threads that are created out of the same thread routine with the same parameters. However, this is a very limiting criterion. Although in practice multiple threads are often created out of the same thread routine, it is quite common that the threads shared the same routine are fed with different parameters for various purposes. Furthermore, light-weight methods such as static analysis cannot infer in general whether the threads are still symmetric after steps of execution, even if threads are spawned from the same routine with the same parameters. Hence, to maximize symmetry reduction, we need to examine the local states of threads during the concrete executions of programs.

As system calls are widely used in multithreaded C programs, it is difficult to precisely capture and compare the local states of threads while both the program under test and the dynamic verifier execute as peer user-level applications. Java PathFinder [11] and SPIN [12] capture the states of multithreaded programs by interpreting the execution. Interpretation requires the model checker to have its own stubs for library routines. As it is difficult or impossible to create stubs for system calls (e.g., file operations) that are widely used among applications written in C, in general we cannot take the interpretation approach to examine multithreaded applications written in C.

In this paper we propose a new algorithm for revealing symmetry in multithreaded programs and show how it can be combined seamlessly with dynamic partial order reduction (DPOR) [4] for more efficient dynamic model checking.

To overcome the state capture problem, we use dynamic program analysis to examine the local state of thread and to discover symmetry in multithreaded programs. That is, instead of statically analyzing the program, we conduct program analysis during the concrete execution of the program.

In more detail, we compute the *residual code* of threads while executing the program. The residual code of a thread τ at a state s is the code that may be executed by τ from s before τ's termination. If we find that the residual code of threads can be put into *syntactic* equivalence under a bijection over thread local variable names and label names, we further check whether the local states of threads can also be put into equivalence under this bijection. We shall prove that, in a state s of a multithreaded program, if the residual code of two threads at s are syntactically equivalent under a certain bijection of thread local variables and labels, and the local states of threads in s are identical under the same bijection, then there is a transition symmetry for the enabled transitions of the two threads at s (details in Section 3). Since transition symmetries induce an equivalence relation on states of the concurrent system, during model checking, we can discard a state s, or backtrack in the context of stateless search, if an equivalent state s' has been explored before. As pointed out in [10], transition symmetry reduction is orthogonal to, and can be combined with partial order reduction techniques for safety verification.

We implemented this symmetry discovery algorithm on top of our dynamic model checker `Inspect` [3,13]. Our experiments shows that our symmetry discovery

algorithm can successfully discover transition symmetries that are hard or otherwise cumbersome to identify manually, and can significantly reduce the checking time for realistic multithreaded applications.

The rest of this paper is organized as follows. In section 2, we review the relevant technical background. In Section 3, we present our algorithm for discovering symmetry in multithreaded programs. In Section 4, we show how to combine the symmetry discovery algorithm with a popular algorithm of dynamic partial order reduction (DPOR). In Section 5 and Section 6, we explain the implementation details and present our experimental results. We review related work in Section 7 and then conclude this paper in Section 8.

2 Preliminaries

2.1 Concurrent Programs

We consider a multithreaded program with a fixed number of sequential threads as a state transition system. We use $Tid = \{1, \ldots, n\}$ to denote the set of thread identities. Threads communicate with each other via *global* objects which are visible to all threads. The operations on global objects are called *visible operations*, while thread local variable updates are *invisible operations*. A *state* of a multithreaded program consists of the global state, and the local state of each thread. The local state of a thread is a stack which is composed of frames. We assign each frame of the stack a unique id. Hence the local state can be viewed as a map from $Fid \subset \mathbb{N}$ to frames. A frame is a map from thread-local variables to the concrete values. Formally, the total system state (S), the local states of all threads ($Locals$), the local state of each thread ($Local$), and the frame of a local state ($Frame$) are defined:

$$\begin{aligned} S &\subseteq Global \times Locals \\ Locals &= Tid \rightarrow Local \\ Local &= Fid \mapsto Frame \\ Frame &= \mathcal{V} \mapsto \mathbb{N} \end{aligned}$$

In this paper, we will assume that the state spaces we consider do not contain any cycles, and focus on detecting deadlocks and safety-property violations such as data races and assertion violations. Note that in model checking of software implementations, acyclic state spaces are quite common. The execution of many practical software applications eventually terminates due to the inputs or the run/test time bound.

For convenience, we use $g(s)$ to denote the global state in a state s, and use $l_\tau(s)$ to denote the local state of thread τ in s. We write $s[\tau := l']$ to denote a state which is identical to s except that the local state of thread τ is l'. We use $s[g := g'; \tau := l']$ to denote a state that is the same as the state s except that the global state is g' and the local state of thread τ is l'.

A transition advances the program from one state to a subsequent state by performing one visible operation of a thread, followed by a finite sequence of invisible operations of the same thread, ending just before the next visible operation of that thread. The transition t_τ of thread τ can be defined as $t_\tau : Global \times Local \rightarrow Global \times Local$.

Let $\mathcal{T}$ denote the set of all transitions of a program. A transition $t_\tau \in \mathcal{T}$ is enabled in a state s if $t_\tau(g(s), l_\tau(s))$ is defined. If t is enabled in s and $t(g, l) = \langle g', l' \rangle$, then

we say the execution of t from s produces a successor state $s' = s[g := g'; \tau := l']$, written $s \xrightarrow{t} s'$.

The behavior of a multithreaded program P is given by a transition system $M = (S, R, s_0)$, where $R \subseteq S \times S$ is the transition relation, and s_0 is the initial state. $(s, s') \in R$ iff $\exists t \in \mathcal{T} : s \xrightarrow{t} s'$.

2.2 A Simple C-Like Programming Language

Taking the source code of a multithreaded program into consideration, a transition t is a sequence of statements $[m_1, m_2, \ldots, m_k]$ of the program, in which the first statement contains a visible operation on the global objects, and the rest of the statements are invisible operations on local objects of the same thread.

To simplify presentation, in this paper, we focus on a C-like simple programming language as shown in Fig. 1. Nevertheless, the theorems we prove in this paper can be extended to accommodate the entire C language. In our actual implementation, we support the entire C language.

$$
\begin{aligned}
P &::= thread^* \\
thread &::= Stmt^* \\
Stmt &::= [l :]s \\
s &::= lhs \leftarrow e \mid \texttt{if } lhs \texttt{ then goto } l' \mid f(params) \mid lhs \leftarrow f(params) \\
params &::= lhs^* \\
lhs &::= v \mid \&v \mid *v \\
e &::= lhs \mid lhs \ \diamond \ lhs \\
&\quad where \diamond \in \{+, -, *, /, \%, <, >, \leq, \geq, \neq, =, ...\}
\end{aligned}
$$

Fig. 1. Syntax of a simple language that is similar to C

As shown in Fig. 1, a program is composed of a set of threads. Each thread is a sequence of statements. A statement can be an assignment, a branch statement, or a function call. A label may be associated with a statement.

We use $\eta(s, t)$ to denote the list of statements that are exercised when a transition t is executed from the state s. We use $VL(\eta(s, t))$ to denote the set of variables and labels that appear in $\eta(s, t)$.

Let $\theta : VL(\eta(s, t)) \rightarrow VL'$ be a mapping from $VL(\eta(s, t))$ to another set of variables and labels VL'. We use $\eta(s, t)[\theta]$ to denote the statement list that we get by replacing each variable and label $v \in VL(\eta(s, t))$ with $\theta(v)$.

Definition 1 (syntactically equivalent transitions). *Let t_1 and t_2 be two transitions of a transition system M that are enabled at state s_1 and s_2 respectively. We say t_1 and t_2 are syntactically equivalent if there exists a bijection $\theta : VL(\eta(s_1, t_1)) \rightarrow VL(\eta(s_2, t_2))$ such that $\eta(s_1, t_1)[\theta] = \eta(s_2, t_2)$.*

2.3 Residual Code of Threads

Let s be a state of a multithreaded program. Let τ be a thread that is enabled in s. The residual code of τ at s is the statements that may be executed by τ from s before its termination. We write $\mathcal{C}(s, \tau)$ to denote this.

The residual code of a thread τ at s captures the code paths that τ may take starting from s. Take the thread shown in Fig. 2(a) as an example. Let the thread be at the point that it finishes line 1 and is going to execute line 2. Assume that the condition at line 2 is false in the state s. If we are able to infer that this condition is false, we can conclude that line 2-5 is the residual code of this thread. Otherwise, we can consider line 1-5, which is an overapproximation of line 2-5, as the residual code.

Fig. 2(b) shows a thread that calls a recursive function. Let the execution context of the thread be $[\texttt{f}(0); \texttt{f}(1)]$, and the thread is in line 5, right before the third call to `f`. Here the parameter `b` has multiple instances in different frames of the call stack, we use b_i to differentiate them. The residual code of the thread at this point is $\langle \texttt{f}(\texttt{b}_2); \texttt{print}(\texttt{b}_2); \texttt{print}(\texttt{b}_1)\rangle$. This can be easily computed by dynamically capturing the execution context of the thread and having an intra-procedural analysis for each function call in the execution context.

```
        thread:
1:   L1: a++;
2:       if (a < 5){
3:         goto L1;
4:       }
5:       local = a;
```

(a) a thread which has a branch statement

```
   thread:
1: f(int b){
2:   if (b > 2)
3:     return;
4:   b = b + 1;
5:   f(b);
6:   print(b);
7: }
```

(b) a thread that starts with $\texttt{f}(0)$

Fig. 2. Examples on the residual code of threads

Let $c = \mathcal{C}(s, \tau)$ be the residual code of thread τ at s. We use $VL(c)$ to denote the set of variables and labels that appears in c. Let $\theta : VL(c) \rightarrow VL'$ be a mapping from $VL(c)$ to another set of variables and labels. We use $c[\theta]$ to denote the code that we get by replacing each occurrence of variables and labels $v \in VL(c)$ with $\theta(v)$. Let $l_\tau(s)$ be a local state of thread τ at s. We use $l_\tau(s)[\theta]$ to denote a new local state we get by renaming each variable v that appears in $l_\tau(s)$ with $\theta(v)$.

Definition 2 (syntactically equivalent residual code). *Let $c_1 = \mathcal{C}(s_1, a)$ and $c_2 = \mathcal{C}(s_2, b)$ be the residual code of thread a at s_1 and thread b at s_2 respectively. We say that c_1 and c_2 are syntactically equivalent if there is a bijection $\theta : VL(c_1) \rightarrow VL(c_2)$ such that $c_1[\theta] = c_2$. We denote this with $c_1 \overset{\theta}{\sim} c_2$.*

2.4 Symmetry

The main idea of symmetry reduction [7,8,9] is that symmetries in the system induce an equivalence relation on states of the concurrent system. While performing model checking, one can discard a state s if an equivalence state s' has been explored before. Here we briefly review the formal definition of symmetry.

Definition 3 (automorphism). *An automorphism on a transition system* $M{=}(S, R, s_0)$ *is a bijection* $\sigma : S \to S$ *such that* $\forall s_1, s_2 \in S : (s_1, s_2) \in R \iff (\sigma(s_1), \sigma(s_2)) \in R$.

Let M be a transition system of a program. Let $\mathbf{id}$ be the identity automorphism on M. For any set $\mathcal{A}$ of automorphism, the closure $G_{\mathcal{A}}$ of $\mathcal{A} \cup \{\mathbf{id}\}$ under inverse and composition is a group. We call such a group *a symmetry group* of M.

The symmetry group $G_{\mathcal{A}}$ induces an equivalence relation $\equiv_{\mathcal{A}}$ on S such that $s_1 \equiv_{\mathcal{A}} s_2$ if $\exists \sigma \in G_{\mathcal{A}} : s_2 = \sigma(s_1)$. The equivalent class $[s] = \{\sigma(s) \mid \sigma \in G_{\mathcal{A}}\}$ of s under $\equiv_{\mathcal{A}}$ is called the *orbit* of s under $G_{\mathcal{A}}$.

Definition 4 (quotient transition system). *Given a transition system* $M = (S, R, s_0)$ *and a symmetry group* $G_{\mathcal{A}}$ *of* M, *a* quotient transition system *for* M *modulo* $G_{\mathcal{A}}$ *is a transition system* $M_{[\mathcal{A}]} = (S', R', s'_0)$ *in which* $S' = \{[s] \mid s \in S\}$, $R' = \{([s], [s']) \mid (s, s') \in S\}$, *and* $s'_0 = [s_0]$.

When we say that there is a symmetry in a program, we mean that there is an automorphism other than the identity automorphism $\mathbf{id}$ on the transition system of the program.

Theorem 1 (reachability). *Given a transition system* $M = (S, R, s_0)$ *with a set of automorphism* $\mathcal{A}$ *on* M, s *is reachable from* s_0 *in* M *if and only if* $[s]$ *is reachable from* $[s_0]$ *in* $M_{[\mathcal{A}]}$.

In practice, the quotient transition system of a system is usually generated on-the-fly using a canonicalization function ζ [9]. Let s be a state. This function maps s to a unique representative $\zeta(s)$ of the equivalence class $[s]$. Whenever a state s is visited, $\zeta[s]$ is stored in memory, e.g., in a hash table.

In dynamic model checking, computing the runtime states of the system precisely is often difficult. In this context, a state is identified by the sequence of transitions that were executed from the initial state s_0 to reach the state. Based on this observation, one can explore symmetry on transitions (*e.g.*, as in [10]) instead of on states.

Definition 5. *Let* $t = (s, s')$ *be a transition of a transition system* M. *Let* σ *denote an automorphism in a symmetry group* $G_{\mathcal{A}}$ *of* M. *We use* $\sigma(t)$ *to denote the transition* $(\sigma(s), \sigma(s'))$. *The relation* $\equiv_{\mathcal{A}}$ *on transitions is defined as:* $t \equiv_{\mathcal{A}} t'$ *if* $\exists \sigma \in G_{\mathcal{A}} : t' = \sigma(t)$.

It is not difficult to prove that the relation $\equiv_{\mathcal{A}}$ on transitions is an equivalence relation. One can extend $\equiv_{\mathcal{A}}$ on transitions to sequences of transitions.

Definition 6. *Let* $w = t_1 t_2 \dots t_n$ *be a nonempty sequence of transitions of a transition system* M. *Let* $\mathcal{A}$ *be a set of automorphism on* M, *and* $G_{\mathcal{A}}$ *be the symmetry group of* $\mathcal{A}$. *Let* $\sigma \in G_{\mathcal{A}}$. *We write* $\sigma(w)$ *to denote the transition sequence* $\sigma(t_1)\sigma(t_2)\dots\sigma(t_n)$. *The relation* $\equiv_{\mathcal{A}}$ *on nonempty sequences of transitions is defined as:* $w \equiv_{\mathcal{A}} w'$ *if* $\exists \sigma \in G_{\mathcal{A}} : w' = \sigma(w)$.

Here $\equiv_{\mathcal{A}}$ is also an equivalence relation. Based on the above definition, we use $[t]$ to denote the equivalence class $[t] = \{\sigma(t) \mid \sigma \in G_{\mathcal{A}}\}$ of t under $\equiv_{\mathcal{A}}$. Similarly, we use $[w] = \{\sigma(w) \mid \sigma \in G_{\mathcal{A}}\}$ to denote the equivalence class of w under $\equiv_{\mathcal{A}}$.

Definition 7 (quotient transition system under $\equiv_{\mathcal{A}}$). *Let $M = (S, R, s_0)$ be a transition system. Let $G_{\mathcal{A}}$ be a symmetry group of M. A quotient transition system for M modulo $G_{\mathcal{A}}$ defined with an equivalence relation $\equiv_{\mathcal{A}}$ on sequences of transitions is a transition system $M_{[\mathcal{A}]} = (S', R', s'_0)$ where $S' = \{[w] \mid s_0 \stackrel{w}{\Rightarrow} s \text{ in } M\}$, $R' = \{([w], [wt]) | s_0 \stackrel{w}{\Rightarrow} s \text{ and } \exists s' \in R : s \stackrel{t}{\rightarrow} s'\}$, and $s'_0 = [\epsilon]$ (the empty word).*

Theorem 2. *Let $M = (S, R, s_0)$ be a transition system and $M_{[\mathcal{A}]} = (S', R', s'_0)$ be a quotient transition system for M modulo a symmetry group $G_{\mathcal{A}}$ of M. Let s be a state in S. s is reachable from s_0 in M via w if and only if $[w]$ is reachable from s'_0 in M'.*

3 Discovering Transition Symmetry

Following the definitions in Section 2, to reveal symmetry in a transition system, we need to find conditions which imply the existence of an automorphism of the transition system. As it is difficult to capture the states of multithreaded programs at runtime, the method of using canonicalization functions to reveal symmetry does not work well here. Our main idea is dynamically analyzing the residual code and the local states of threads to discover symmetry.

Let s be a state in the transition systems, and τ be a thread that is enabled in a state s. In a concrete execution, the transition that τ can execute from s is determined by the residual code $\mathcal{C}(s, \tau)$, the global state $g(s)$, and the local state of τ in s, *i.e.*, $l_\tau(s)$. In this section, we present a simple algorithm based on this idea.

Let t_a and t_b be two transitions that are enabled at s by thread a and b, and let $c_a = \mathcal{C}(s, a)$ and $c_b = \mathcal{C}(s, b)$ be the residual code of thread a and b at s respectively. In our algorithm, first we try to construct a bijection $\theta : VL(c_a) \rightarrow VL(c_b)$ such that c_a and c_b are syntactically equivalent under θ, that is, $c_a \stackrel{\theta}{\sim} c_b$. If such a θ can be constructed, we check whether the local states of threads are equivalent under θ. We can prove that transitions t_a and t_b are symmetric if the c_1 and c_2 are syntactically equivalent, and the local state of thread are equivalent under θ (detailed proof is in Section 3.3).

3.1 Inferring Syntactic Equivalence among Residual Code

Let c_a and c_b be residual code of threads a and b that follow the syntax in Fig. 1. Fig. 3 shows the rules we use to construct a bijection θ from $VL(c_a)$ to $VL(c_b)$. We only conclude that c_a and c_b are syntactically equivalent if such a bijection can be constructed by successfully applying these rules. Otherwise, we treat c_a and c_b as non-syntactic-equivalent.

In these rules, we use m^τ to denote a statement in the residual code of thread τ, l^τ to denote a label in the residual code of thread τ. We use v_g to denote a global variable, v^τ to denote a local variable of thread τ, and c to denote a constant.

To simplify the presentation, we assume that the residual code c_a and c_b are two lists of statements. In practice, the residual code can be presented as a control flow graph or an abstract syntax tree. We can easily extend our rules to handle these structures. In our implementation, we represent the residual code of a thread as a control flow graph.

We start by applying rule R0 to c_a and c_b. R0 first checks whether m^a_1, which is the first statement in c_a, is syntactically equivalent to m^b_1, which is the first statement

$$\frac{(m_1^a,\ m_1^b) \vdash \theta_1;\quad ([m_2^a,\ldots,m_n^a],\ [m_2^b,\ldots,m_n^b]) \vdash \theta_2}{([m_1^a,m_2^a,\ldots,m_n^a],\ [m_1^b,m_2^b,\ldots,m_n^b]) \vdash \theta_1 \cup \theta_2}\ \text{R0}$$

$$\frac{(m^a,\ m^b) \vdash \theta;\quad (l^a,\ l^b) \vdash [l^a \mapsto l^b]}{(l^a : m^a,\ l^b : m^b) \vdash \theta \cup [l^a \mapsto l^b]}\ \text{R1}$$

$$\frac{(e^a,\ e^b) \vdash \theta;\quad (l^a,\ l^b) \vdash [l^a \mapsto l^b]}{(\texttt{if } e^a \texttt{ then goto } l^a,\quad \texttt{if } e^b \texttt{ then goto } l^b) \vdash \theta \cup [l^a \mapsto l^b]}\ \text{R2}$$

$$\frac{(h^a,\ h^b) \vdash \theta_1;\quad (e^a,\ e^b) \vdash \theta_2}{(h^a = e^a,\ h^b = e^b) \vdash \theta_1 \cup \theta_2}\ \text{R3} \qquad \frac{(f(p_1,\ldots,p_n),\ f(q_1,\ldots,q_n)) \vdash \theta_1;\quad (h^a,\ h^b) \vdash \theta_2}{(h^a = f(p_1,\ldots,p_n),\ h^b = f(q_1,\ldots,q_n)) \vdash \theta_1 \cup \theta_2}\ \text{R4}$$

$$\frac{(h_1^a,\ h_1^b) \vdash \theta_1 \quad (h_2^a,\ h_2^b) \vdash \theta_2}{(h_1^a \diamond h_2^a,\ h_1^b \diamond h_2^b) \vdash \theta_1 \cup \theta_2}\ \text{R5} \qquad \frac{([p_1,\ldots,p_n],\ [q_1,\ldots,q_n]) \vdash \theta}{(f(p_1,\ldots,p_n),\ f(q_1,\ldots,q_n)) \vdash \theta}\ \text{R6}$$

$$\frac{}{(v_g,\ v_g) \vdash [v_g \mapsto v_g]}\ \text{R7} \qquad \frac{}{(\&v_g,\ \&v_g) \vdash [v_g \mapsto v_g]}\ \text{R8} \qquad \frac{}{(*v_g,\ *v_g) \vdash [v_g \mapsto v_g]}\ \text{R9}$$

$$\frac{}{(\&v^a,\ \&v^b) \vdash [v^a \mapsto v^b]}\ \text{R10} \qquad \frac{}{(*v^a,\ *v^b) \vdash [v^a \mapsto v^b]}\ \text{R11} \qquad \frac{}{(v^a,\ v^b) \vdash [v^a \mapsto v^b]}\ \text{R12}$$

$$\frac{}{(c,\ c) \vdash [\,]}\ \text{R13}$$

[]

Fig. 3. Rules for inferring syntactic equivalence among residual code of threads

of c_b, by recursively applying other rules. Next, R0 recursively checks whether the rest of c_a are syntactically equivalent to the rest of c_b. If m_1^a and m_1^b are syntactically equivalent under θ_1, and the rest of residual code are syntactically equivalent under θ_2, we conclude that the two statement lists are syntactically equivalent under $\theta_1 \cup \theta_2$.

Rules R1 - R4 handle different kinds of statements, and rules R5 - R13 handle different forms of expressions. Specifically, rules R7 - R9 guarantee that the bijection we construct only maps a global variable to itself. We can also extend the rules to be more semantic-aware, for instance, by taking commutativity of binary operators into consideration (e.g., $x + y$ versus $y + x$).

Theorem 3. *Let $c_1 = \mathcal{C}(s_1, a)$ and $c_2 = \mathcal{C}(s_2, b)$ be the residual code of thread a at s_1 and b at s_b respectively. If a bijection $\theta : VL(c_1) \rightarrow VL(c_2)$ between c_1 and c_2 can be constructed following the rules in Fig. 3, c_1 and c_2 are syntactically equivalent.*

Theorem 3 can be easily proved by induction on the length of the statements.

3.2 Discovering Symmetric Transitions

Syntactic equivalence between the residual code of threads alone does not imply transition symmetry, as different threads may take different paths depending on the local states of threads. To soundly infer transition symmetry, we also need to examine the local states of threads. The procedure SYMMETRIC in Fig. 4 shows our algorithm for detecting transition symmetry.

```
1: SYMMETRIC(s, a, b) {
2:     let c1 = C(s, a) and c2 = C(s, b) ;
3:     if ( following the rules in Fig. 3, there is a bijection θ such that C(s, a) ~θ C(s, b) )
4:         return l_a(s)[θ] = l_b(s);
5:     else
6:         return FALSE;
7: }
```

Fig. 4. Checking whether two transitions enabled at s by a and b are symmetric

SYMMETRIC accepts three parameters as inputs: a state s and a pair of threads that are enabled in s. It returns TRUE or FALSE. Let a and b be the two threads. To test whether the transitions that enabled by a and b at s are symmetric transitions, we first compute the residual code of a and b (line 2 or Fig. 4), which are c_1 and c_2 respectively. Then we check whether a bijection between $VL(c_1)$ and $VL(c_2)$ can be constructed following the rules in Fig. 3. If such a bijection cannot be constructed, SYMMETRIC returns FALSE. Otherwise, let θ be the constructed bijection, we check whether the local states of a and b at s are equivalent under θ (line 4 of Fig. 4). We only conclude transition symmetry if the local states of threads $l_a(s)$ and $l_b(s)$ are equivalent under this bijection.

3.3 Soundness

Lemma 1. *Let $c_a = \mathcal{C}(s_a, a)$ and $c_b = \mathcal{C}(s_b, b)$ be the residual code of thread a at s_a and thread b at s_b respectively. Let t_a be a transition that is enabled at s_a by thread a such that $s_a \xrightarrow{t_a} s'_a$. Suppose we can construct $\theta : VL(c_a) \to VL(c_b)$ following the rules in Fig. 3 such that $c_a \overset{\theta}{\sim} c_b$. Now, if we have $g(s_a) = g(s_b)$, $l_a(s_a)[\theta] = l_b(s_b)$, there must exist a transition t_b that is enabled by thread b in s_b such that $s_b \xrightarrow{t_b} s'_b$, $g(s'_b) = g(s'_a)$, and $l_a(s'_a)[\theta] = l_b(s'_b)$.*

Proof. A transition t of thread τ that is enabled at a state s is determined by the global state $g(s)$, the local state $l_\tau(s)$, and the residual code of τ. Let $\eta(s_a, t_a)$, which is the list of statements to be exercised while executing t_a from s_a, be $[m_1, \ldots, m_k]$. As $g(s_a) = g(s_b)$, $l_a(s_a)[\theta] = l_b(s_b)$, and we can construct a bijection θ following Fig. 3 such that $c_a \overset{\theta}{\sim} c_b$, we have $\eta(s_b, t_b) = \eta(s_a, t_a)[\theta]$ (This can be proved by induction on k. We omit the details here.). Obviously there is s'_b such that $s_b \xrightarrow{t_b} s'_b$.

As $t_a \overset{\theta}{\sim} t_b$ and θ is constructed following the rules in Fig. 3, $\eta(s_a, t_a)$ and $\eta(s_b, t_b)$ must have the same visible operation. As only the visible operations may update global objects, we have $g(s'_a) = g(s'_b)$.

$l_a(s'_a)$ is determined by $g(s'_a)$, $l_a(s_a)$, and the invisible operations in t_a, and $l_b(s'_b)$ is determined by $g(s'_b)$, $l_b(s_b)$, and the invisible operations in t_b. As $t_a \overset{\theta}{\sim} t_b$, we can use induction to prove that if $l_a(s_a)[\theta] = l_b(s_b)$, $l_a(s'_a)[\theta] = l_b(s'_b)$. □

Lemma 2. *Let t_a and t_b be two transitions that are enabled at a state s by thread a and b, and let s_a and s_b be two states in M such that $s \xrightarrow{t_a} s_a$ and $s \xrightarrow{t_b} s_b$. If* SYMMETRIC

(s, a, b) returns TRUE*, let* $\theta : VL(\mathcal{C}(s,a)) \rightarrow VL(\mathcal{C}(s,b))$ *be the bijection that we construct following the rules in Fig. 3, we have* $s_b = s_a[a := l_b(s_a)[\theta^{-1}]; b := l_a(s_a)[\theta]]$.

Proof. As a transition t that is enabled at a state s can only changes the global states of s and the local state of $tid(t)$ in s, t_a does not change the local state of b in s. Hence, $l_b(s_a) = l_b(s)$. Similarly, we have $l_a(s_b) = l_a(s)$. According to Lemma 1, we have $g(s_b) = g(s_a)$, and $l_b(s_b) = l_a(s_a)[\theta]$. Based on this, we have

$$\begin{aligned} s_b &= s_a[a := l_a(s); b := l_b(s_b)] = s_a[a := l_a(s)[\theta][\theta^{-1}]; b := l_b(s_b)] \\ &= s_a[a := l_b(s)[\theta^{-1}]; b := l_b(s_b)] = s_a[a := l_b(s_a)[\theta^{-1}]; b := l_a(s_a)[\theta]] \end{aligned}$$

□

Lemma 3. *Let* t_a *and* t_b *be two transitions that are enabled at a state* s *by thread* a *and* b*. If* SYMMETRIC*(s, a, b) returns* TRUE*, let* θ *be the bijection from* $VL(\mathcal{C}(s,a))$ *to* $VL(\mathcal{C}(s,b))$ *that we construct following the rules in Fig. 3. Then for any transition sequence* $t_1, \ldots, t_k$ *in* M *such that* $s_c \xrightarrow{t_a} s_a \xrightarrow{t_1} s_1 \ldots \xrightarrow{t_k} s_k$*, there must exist another transition sequence* $t'_1, \ldots, t'_k$ *in* M *such that* $s_c \xrightarrow{t_b} s_b \xrightarrow{t'_1} s'_1 \ldots \xrightarrow{t'_k} s'_k$ *and for all* i*,* $1 \leq i \leq k$*,* $s'_i = s_i[a := l_b(s_i)[\theta^{-1}]; b := l_a(s_i)[\theta]]$.

Here we only provide a sketch of the proof here. We can prove this lemma by induction on the length of the transition sequence. In the base case, $k = 1$. Let t_1 be a transition such that $s \xrightarrow{t_a} s_a \xrightarrow{t_1} s_1$. Following Lemma 2, we have $g(s_a) = g(s_b)$, and $l_b(s_b) = l_a(s_a)[\theta]$. Based on this and Lemma 1, we can prove the base case by studying three cases: (i) $tid(t_1) \neq a$ and $tid(t_1) \neq b$; (ii) $tid(t_1) = a$ and (iii) $tid(t_1) = b$. The induction step can be proved similarly.

Theorem 4. *Let* s_m *be a state of a transition system* $M = (S, R, s_0)$*. Let* t_a *and* t_b *be two transitions that are enabled at* s_m *by thread* a *and* b*. If* SYMMETRIC*(*s_m*, a, b) returns* TRUE*,* t_a *and* t_b *are symmetric transitions.*

Proof. Let $c_1 = \mathcal{C}(s_m, a)$ and $c_2 = \mathcal{C}(s_m, b)$ be the residual code of a and b at the state s. Let t_a and t_b be the transitions of a and b at s such that $s_m \xrightarrow{t_a} s_a$ and $s_m \xrightarrow{t_b} s_b$. If SYMMETRIC($s_m$, a, b) returns TRUE, let $\theta : VL(c_1) \rightarrow VL(c_2)$ be the bijection that we construct following the rules in Fig. 3. We can construct an automorphism σ on M as follows:

$$\sigma(s) = \begin{cases} s & \text{if s is not reachable from } s_a \text{ or } s_b \\ s[a := l_b(s)[\theta^{-1}]; b := l_a(s)[\theta]] & \text{otherwise} \end{cases}$$

Obviously, for any state s that is not reachable from s_a or s_b, $(s, s') \in R \iff (\sigma(s), \sigma(s')) \in R$ holds.

For any state s_k that is reachable from s_a or s_b. Let $s_a \xrightarrow{t_1} s_1 \ldots \xrightarrow{t_k} s_k$ be the transition sequence from s_a to reach s_k. Following Lemma 3, if $(s_k, s_{k+1}) \in R$, $(\sigma(s_k), \sigma(s_{k+1})) \in R$. If $(\sigma(s_k), \sigma(s_{k+1})) \in R$, with $\theta' = \theta^{-1}$, again according to Lemma 3, we have $(s_k, s_{k+1}) \in R$. Hence σ is an automorphism of M. □

4 Dynamic Partial Order Reduction with Symmetry Discovery

Dynamic partial order reduction (DPOR) [4] is an effective algorithm for reducing redundant interleavings in dynamic model checking. In DPOR, given a state s, the persistent set of s [14] is not computed immediately after reaching s. Instead, DPOR explores the states that can be reached from s using depth-first search, and dynamically adds backtrack information into the backtrack set of s while exploring the sub-space that is reachable from s.

In more detail, let t_i be a transition that is enabled at state s. Suppose the model checker first selects t to execute at s. Let t_j be a transition which can be enabled with a depth first search (in one or more steps) from s by executing t_i. Then before t_j is executed, DPOR will check whether t_j and t_i are dependent and can be enabled concurrently. If so, $tid(t_j)$ or the id of the thread which t_j is dependent on will be added

```
1: Initially: S is empty; SymDpor(S, s_0)

2: SymDpor(S, s) {
3:       S.push(s);
4:       for each t ∈ s.enabled, UpdateBacktrackSets(S, t);
5:       let τ ∈ Tid such that ∃t ∈ s.enabled : tid(t) = τ;
6:       s.backtrack ← {τ};
7:       s.done ← ∅;
8:       while (∃t ∈ s.backtrack \ s.done) {
9:          s.done ← s.done ∪ {t};
10:         s.backtrack ← s.backtrack \ {t};
11:         if (∀t' ∈ s.done : ¬Symmetric(s, tid(t), tid(t'))){
12:            let s' ∈ S such that s -t-> s';
13:            SymDpor(S, s');
14:            S.pop(s);
15:         }
16:      }
17: }

18: UpdateBacktrackSets(S, t) {
19:    let T be the sequence of transitions associated with S;
20:    let t_d be the latest transition in T that is dependent and may be co-enabled with t;
21:    if (t_d ≠ null){
22:       let s_d be the state in S from which t_d is executed;
23:       let E be {q ∈ s_d.enabled | tid(q) = tid(t), or q in T, q was after t_d in T,
             and there is a happens-before relation for (q, t)}
24:       if (E ≠ ∅)
25:          choose any q in E, add tid(q) to s_d.backtrack;
26:       else
27:          s_d.backtrack ← s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
28:    }
29: }
```

Fig. 5. Dynamic partial order reduction with symmetry discovery

to the backtrack set of s if a transition of $tid(t_j)$ is enabled at s. Later, in the process of backtracking, if the state s is found with non-empty backtrack set, DPOR will select one transition t which is enabled at s and $tid(t)$ is in the backtrack set of s, and explore a new branch of the state space by executing t from s; at the same time, $tid(t)$ will be removed from the backtrack set of s.

As shown in Fig. 5, combining our transition symmetry discovery method with dynamic partial order reduction is straightforward. In this algorithm, we use $s.enabled$ to denote the set of enabled transitions in a state s, $s.backtrack$ to denote the set of enabled threads that need to be explored at a state s, and $s.done$ to denote the set of enabled threads the transitions of which have been executed at s. Comparing with the original DPOR algorithm, the only place we need to change w.r.t. the original DPOR algorithm is in line 11 of Fig. 5, where we check whether a symmetric transition has been explored before exploring a transition.

Theorem 5. *Consider a terminating multithreaded program M. If there exist deadlocks in the state space of M,* SYMDPOR *will visit at least one of them. If there exist data races in the state space of M,* SYMDPOR *will visit at least one of them. If we encode local assertions as part of the residual code, and there exist local assertion violations in the state space of M,* SYMDPOR *will visit at least one of them.*

Proof. Following the definition of deadlock, data races, or local assertion violations, and Theorem 4, if a state s has a deadlock , each state s' in $[s]$ have a deadlock. Thus exploring one state per equivalent class $[s]$ shall be sufficient to detect deadlocks in multithreaded programs. Similarly, the theorem holds for data races. As for local assertions, if they are encoded as part of the residual code and there exist local assertion violations in the program, all states that belong to the same equivalent class must violate the assertion. Hence the theorem holds.

5 Implementation

We have implemented the transition symmetry discovery algorithm on top of our dynamic model checker `Inspect` [3,13]. The workflow of `Inspect` with automatic symmetry discovery is shown in Figure 6. It consists of four parts: (i) a program analyzer that analyzes the program for possible global accesses and other information of the program, (ii) a program instrumentor that can instrument the program at the source code level with codes that are used to communicate with the scheduler, etc., (iii) a thread library wrapper that helps intercept the thread library calls, and (iv) an external scheduler that schedules the interleaved executions of the threads.

To implement the symmetry discovery algorithm, we enhance the scheduler to make it capable of computing the residual code of threads. Besides, we inject a prober thread into the program under test to make it possible for the scheduler to examine the execution contexts and the local states of threads during the concrete execution of a program.

The idea behind the prober thread is that, as all threads in a process share the same address space, we can use a prober thread to get the values and addresses of variables in other threads, as well as the execution contexts of threads. Figure 7 shows the routine of the prober thread. The task of a prober thread is straightforward: it keeps passively

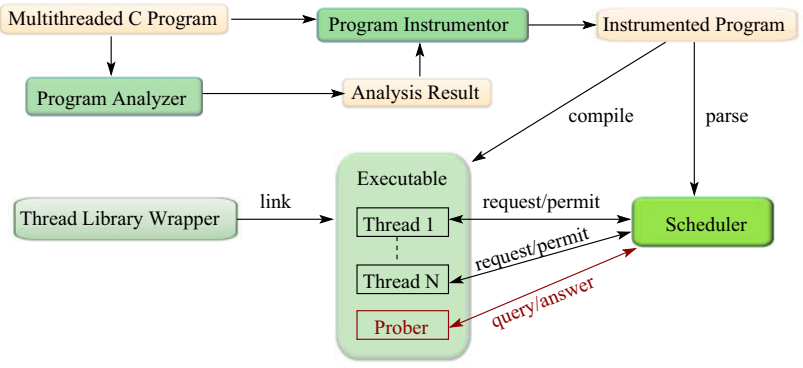

Fig. 6. The workflow of `Inspect` with automatic symmetry discovery

```
1: while (TRUE) {
2:     wait a query request from the scheduler;
3:     probe the value of a variable or the execution context of a thread according to the request;
4:     send the result to the scheduler;
5: }
```

Fig. 7. The procedure of the probing thread

waiting for query requests from the external scheduler, act accordingly, and send results back to the scheduler.

To examine the local variables of other threads, the prober thread needs an index of local variables of threads with which it can locate specific variables. We improve the program instrumentor to have it be able to instrument code that can help to build such an index. Besides the instrumentation presented in [3], the program instrumentor is required to (i) add code right before the first statement of each function to register local variables in the new stack frame, (ii) add code right before the return statement of each function to remove local variables of the current stack frame from the index, and (iii) add struct interpretation functions for user-defined structs with which the prober can access specific fields of the structs. We use CIL [15] as the front end for parsing and instrumenting the multithreaded C programs.

To compute the residual code of threads, before starting model checking, the scheduler parse the program and construct a control flow graph for every function. To compute the residual code $\mathcal{C}(s, \tau)$ of thread τ at a state s during the search, first the scheduler gets the execution context of τ with the help of the prober thread. Then, for each function in the execution context, the scheduler conducts an intra-procedural analysis to get an AST which represents the statements that may be executed by τ before it returns from the function. In our implementation, $\mathcal{C}(s, \tau)$ is represented as a list of ASTs that we get by examining every function in the execution context.

6 Experimental Results

We conducted experiments on two realistic multithreaded benchmarks, `aget` and `pfscan`. `aget` [16] is an ftp client which uses multiple worker threads to download different segments of a large file concurrently. `pfscan` [17] is a multithreaded file scanner that uses multiple threads to search in parallel through directories. `aget-buggy` and `pfscan-buggy` are buggy versions of `aget` and `pfscan` with inserted data race and deadlock bugs. All benchmarks are accompanied by test cases to facilitate the concrete execution. Our experiments were conducted on a workstation with 2.8 GHz Pentium D processor and 2GB memory running Fedora 5.

Table 1 compares the results of checking the set of benchmarks using dynamic partial order reduction (DPOR) and SYMDPOR. The first three columns show the statistics of the test cases, including the name, the line of code, and the number of threads. Columns 4-5 compare the two methods in terms of their total runtime in seconds. Columns 6-7 compare the number of explored executions before they produce verification results. Column 8-9 compare the number of explored transitions. As shown in the table, our symmetry discovery scheme can help to significantly reduce the checking time, with only modest overheads. For instance, the symmetry checking adds 15%-40% overhead on the time per execution of `pfscan`. However, this overhead is well compensated by the checking time we save with symmetry reduction. Furthermore, the symmetry discovery step can be made more efficient by precomputing the bijections between residual code of threads and storing them in a hash table along with the bijections.

Table 2 shows the overhead of dynamic analysis while checking `pfscan` and `pfscan-buggy` using SYMDPOR. In this table, the first three columns shows the benchmark, the number of threads and the number of executions for the checking. Column 4 (Total) shows the total checking time. Column 5 (Probing) shows the time that was spent on communcating with the prober thread to learn the local state of threads. Column 6 (Residual+Bijection) shows the time for computing the residual code of threads and constructing the bijection between local variables and labels of threads. Column 7 (Analysis) shows the number of times that SYMMETRIC is called. Column 8 (Success) shows the number of times that SYMMETRIC returns TRUE. The time that *directly* spent on SYMMETRIC includes probing time and bijection computation time.

Table 1. Comparing DPOR and SYMDPOR on checking two benchmarks

Test programs			Runtime(s)		Executions		Transitions	
Benchmark	loc	thrds	DPOR	SymDpor	DPOR	SymDpor	DPOR	SymDpor
aget-buggy	1233	4	29107	16	1009010	420	30233023	19084
aget-buggy	1233	5	$>$ 86400	37	-	926	-	32485
aget	1233	4	$>$ 86400	18	-	462	-	16263
aget	1233	5	$>$ 86400	258	-	6006	-	211256
aget	1233	6	$>$ 86400	2579	-	87516	-	3117152
pfscan-buggy	921	3	2	1	120	71	1980	1206
pfscan-buggy	921	4	389	50	28079	3148	428410	50296
pfscan	921	3	21	3	1096	136	18006	2334
pfscan	921	4	64067	155	4184546	7111	64465088	85074
pfscan	921	5	$>$ 86400	5147	-	322695	-	5373766

Table 2. Analysis on the overhead of dynamic analysis

Benchmark	Threads	Executions	Time (sec)			Dynamic Analysis	
			Total	Probing	Residual + Bijection	Analysis	Success
pfscan-buggy	3	71	1	0.05	0.01	103	29
pfscan-buggy	4	3148	50	0.35	0.04	2613	230
pfscan	3	136	3.18	0.04	0.02	207	51
pfscan	4	7111	155.3	1.92	0.48	11326	3275
pfscan	5	322695	5147	81.41	20.03	1685733	544816

The results show that probing the local states of threads, computing residual code of threads, and constructing bijections among local variables only cost a small fraction ($< 2\%$) of the total checking time. Most of the 15%-40% slowdown per execution is contributed by the code that is instrumented for supporting dynamic analysis.

7 Related Work

There has been a lot of research on automatic symmetry discovery. In solving boolean satisfiability, a typical approach is to convert the problem into a graph and employ graph symmetry tool to uncover symmetry [18]. Another approach for discovering symmetry is boolean matching [19], which converts the boolean constraints into a canonization form to reveal symmetries. In domains such as microprocessor verification, the graph often has a large number of vertices, however, the average number of neighbors of a vertex is usually small. Several algorithms based on exploiting this fact [20,21] are proposed to efficiently handle these graphs. More recent effort on discovery symmetry using sparsity [22] significantly reduced the discovery time by exploiting the sparsity in both the input and the output of the system.

In explicit state model checking, adaptive symmetry reduction [23] has been proposed to dynamically discover symmetry in a concurrent system on the fly. This is close in spirit to our work. [23] introduces the notion of subsumption, which means that a state subsumes another if its orbit contains that of the other one. Subsumption induces a quotient structure with an equivalent set of reachable states. However, [23] did not address the practical problems for discovering symmetries in multithreaded programs to improve the efficiency of dynamic verification. Our algorithm can revealing symmetries in realistic multithreaded programs. We have proven this with an efficient practical implementation.

In software model checking, state canonicalization has been the primary method to reveal symmetry. Efficient canonization functions [24,25,26,27] have been proposed to handle heap symmetry in Java programs which create objects in dynamic area. As these algorithms assume that the model checker is capable of capturing the states of concurrent programs, we cannot utilize them in dynamic verification to reveal symmetries.

In dynamic model checking of concurrent programs, transition symmetry [10] has been the main method for exploiting symmetry at the whole process level. However, in [10], the user is required to come up with a permutation function, which is then used by the algorithm to check whether two transitions are symmetric. In practice, it is often difficult to manually specify such a permutation function. By employing dynamic

analysis, our approach automates symmetry discovery. To the best of our knowledge, our algorithm is the first effort in automating symmetry discovery for dynamic model checking.

8 Conclusion and Future Work

We propose a new algorithm that uses dynamic program analysis to discover symmetry in mulithreaded programs. The new algorithm can be easily combined with partial order reduction algorithms and significantly reduce the runtime of dynamic model checking. In future work, we would like to further improve the symmetry discovery algorithm with a more semantic-aware dynamic analysis. Since dynamic analysis can be a helpful technique for testing and verification in many contexts, we are investigating several possibilities in this direction.

References

1. Godefroid, P.: Model Checking for Programming Languages using Verisoft. In: POPL, pp. 174–186 (1997)
2. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 446–455. ACM, New York (2007)
3. Yang, Y., Chen, X., Gopalakrishnan, G.: Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah (2008)
4. Flanagan, C., Godefroid, P.: Dynamic Partial-order Reduction for Model Checking Software. In: Palsberg, J., Abadi, M. (eds.) POPL, pp. 110–121. ACM, New York (2005)
5. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient stateful dynamic partial order reduction. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 288–305. Springer, Heidelberg (2008)
6. Wang, C., Yang, Y., Gupta, A., Gopalakrishnan, G.: Dynamic model checking with property driven pruning to detect race conditions. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 126–140. Springer, Heidelberg (2008)
7. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Form. Methods Syst. Des. 9(1-2), 77–104 (1996)
8. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Form. Methods Syst. Des. 9(1-2), 105–131 (1996)
9. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design 9(1/2), 41–75 (1996)
10. Godefroid, P.: Exploiting symmetry when model-checking software. In: FORTE. IFIP Conference Proceedings, vol. 156, pp. 257–275. Kluwer, Dordrecht (1999)
11. Havelund, K., Pressburger, T.: Model Checking Java Programs using Java PathFinder. STTT 2(4), 366–381 (2000)
12. Zaks, A., Joshi, R.: Verifying multi-threaded c programs with SPIN. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 325–342. Springer, Heidelberg (2008)
13. `http://www.cs.utah.edu/~yuyang/inspect/`
14. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer, Heidelberg (1996)

15. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
16. http://freshmeat.net/projects/aget/
17. http://freshmeat.net/projects/pfscan
18. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult SAT instances in the presence of symmetry. In: DAC, pp. 731–736. ACM, New York (2002)
19. Chai, D., Kuehlmann, A.: Building a better boolean matcher and symmetry detector. In: DATE, pp. 1079–1084 (2006)
20. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for CNF. In: DAC, pp. 530–534. ACM, New York (2004)
21. Junttila, T., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: SIMA Workshop on Algorithm Engineering and Experiments (2007)
22. Darga, P.T., Sakallah, K.A., Markov, I.L.: Faster symmetry discovery using sparsity of symmetries. In: DAC, pp. 149–154. ACM, New York (2008)
23. Wahl, T.: Adaptive symmetry reduction. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 393–405. Springer, Heidelberg (2007)
24. Lerda, F., Visser, W.: Addressing dynamic issues of program model checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 80–102. Springer, Heidelberg (2001)
25. Iosif, R.: Exploiting heap symmetries in explicit-state model checking of software. In: 16th IEEE International Conference on Automated Software Engineering (ASE 2001), Coronado Island, San Diego, CA, USA, November 26-29, 2001, pp. 254–261. IEEE Computer Society, Los Alamitos (2001)
26. Iosif, R.: Symmetry reductions for model checking of concurrent dynamic software. STTT 6(4), 302–319 (2004)
27. Visser, W., Pasareanu, C.S., Pelánek, R.: Test input generation for java containers using state matching. In: Pollock, L.L., Pezzè, M. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006, pp. 37–48. ACM, New York (2006)