Reduction for Compositional Verification of Multi-Threaded Programs

Corneliu Popeea
Technische Universität München

Andrey Rybalchenko
Microsoft Research Cambridge and
Technische Universität München

Andreas Wilhelm
Technische Universität München

***Abstract*—Automated verification of multi-threaded programs requires keeping track of a very large number of possible interactions between the program threads. Different reasoning methods have been proposed that alleviate the explicit enumeration of all thread interleavings, e.g., Lipton's theory of reduction or Owicki-Gries method for compositional reasoning, however their synergistic interplay has not yet been fully explored. In this paper we explore the applicability of the theory of reduction for pruning of equivalent interleavings for the automated verification of multi-threaded programs with infinite-state spaces. We propose proof rules for safety and termination of multi-threaded programs that integrate into an Owicki-Gries based compositional verifier. The verification conditions of our method are Horn clauses, thus facilitating automation by using off-the-shelf Horn clause solvers. We present preliminary experimental results that show the advantages of our approach when compared to state-of-the-art verifiers of C programs.**

I. Introduction

Development of practical verification tools for multi-threaded programs requires dealing with the explosion of the number of thread interleavings that need to be taken into consideration. However, while one can easily construct a contrived program in which every interleaving leads to a different outcome, often enough different interleavings produce equal outcome, and hence can be considered equivalent. Such an equivalence between interleavings suggests that only representatives of each equivalence class need to be considered when verifying a multi-threaded program.

One way to exploit such equivalence is called partial order reduction (POR) [1]. This technique is used in combination with model checking and amounts to restricting the successor computation to representative interleavings, which is performed on-the-fly during the exploration of the model. Explicit-state [1], [2] as well as symbolic [3], [4] model checking algorithms can be effectively combined with POR. Furthermore, recent work shows that POR can also boost interpolation based verification [5], which makes it applicable for the verification of programs with infinite-state spaces.

Alternatively, one can exploit equivalence by transforming a multi-threaded program such that it only produces representative interleavings, or a sufficiently small superset thereof. Such transformation summarises and replaces certain sequences of statements within threads by their composition into so-called *reducible blocks*. Following Lipton's theory of reduction [6], executing these code blocks without any preemption produces representative interleaving when their building blocks commute. Reducible blocks can greatly simplify deductive verification of multi-threaded programs using proof assistants, see e.g. [7]. For finite state systems, reducible blocks (also called transactions in the literature) can be effectively identified and created on-the-fly during model checking [8], [9], [10]. Unfortunately, this does not benefit automatic verification tools for multi-threaded programs with infinite-state spaces. In particular, if reducible blocks contain loops then their invariants are required to replace reducible blocks by their summaries.

In this paper we explore the applicability of reduction for pruning of equivalent interleavings for the automated verification of multi-threaded programs with infinite-state spaces. Since reducible blocks rarely contain all statements of a thread, i.e., there are multiple reducible blocks in each thread as well as some statements that do not belong to any reducible block, we integrate compositional reasoning into our exploration as a complementary technique for avoiding the explicit exploration of all interleavings. That is, our method relies on reduction whenever possible, while statements outside of reducible blocks are subject to compositional reasoning.

Technically, our paper makes the following contributions: 1) a Horn constraint based method for identifying commutativity (mover annotations) of program statements, 2) compositional proof rules for safety and termination that integrate reduction and Owicki-Gries reasoning [11], 3) an efficient implementation based on these ingredients. Our design decisions were directed by the following considerations. Commutativity inference, the first building block of our approach, serves as a preliminary step for a final constraint based verification run. We allow it to be more precise and data dependent in comparison with type based approaches, e.g. [12]. Even though being potentially more expensive, the ability to infer larger transactions at step 1 may lead to dramatic reduction in verification time when dealing with step 2. Our proof rules are also inspired by the use of procedure summaries, see e.g. [13], however instead of being driven by calls/returns to mark start/finish points of summaries, we use transitions that enter/exit from reducible blocks. Note that loops can be part of reducible blocks and summarisation constraints defer reasoning about them to the final solving step.

In summary, this paper shows that reducible blocks can be identified without requiring deep and intricate modification of the underlying verification techniques. Our experimental evaluation shows that the conceptual separation of concerns, i.e., treatment of equivalence between interleavings via reducible blocks and keeping track of interleavings using compositional proof system, compares favourably with state-of-the-art verification approaches.

ISBN: 978-0-9835678-4-4.

```
int x=2, y=2, mx=0, my=0;
// Thread-1
    int a;
 0: acquire(mx);
 1: a = x;
 2: acquire(my);
 3: y = y+a;
 4: release(my);
 5: a = a+1;
 6: acquire(my);
 7: y = y+a;
 8: release(my);
 9: x = 2*x+a;
10: release(mx);
11:

 // Thread-2
 0: acquire(mx);
 1: x = x+2;
 2: release(mx);
 3:

 // Thread-3
 0: acquire(my);
 1: y = y+2;
 2: release(my);
 3:
```

$V_G = (\mathtt{x}, \mathtt{y}, \mathtt{mx}, \mathtt{my}),\ V_1 = (\mathtt{a}, \mathtt{pc}_1),\ V_2 = (\mathtt{pc}_2),\ V_3 = (\mathtt{pc}_3)$

$init(V) = (\mathtt{x} = 2 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0 \wedge \mathtt{pc}_1 = \mathtt{pc}_2 = \mathtt{pc}_3 = \ell_0)$

$next_1(VV') =$
$(move_1(\ell_0, \ell_1) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{my}, \mathtt{a})) \vee$
$(move_1(\ell_1, \ell_2) \wedge \mathtt{a}' = \mathtt{x} \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{mx}, \mathtt{my})) \vee$
$(move_1(\ell_2, \ell_3) \wedge \mathtt{my} = 0 \wedge \mathtt{my}' = 1 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{mx}, \mathtt{a})) \vee$
$(move_1(\ell_3, \ell_4) \wedge \mathtt{y}' = \mathtt{y} + \mathtt{a} \wedge skip(\mathtt{x}, \mathtt{mx}, \mathtt{my}, \mathtt{a})) \vee$
$(move_1(\ell_4, \ell_5) \wedge \mathtt{my}' = 0 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{mx}, \mathtt{a})) \vee$
$(move_1(\ell_5, \ell_6) \wedge \mathtt{a}' = \mathtt{a} + 1 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{mx}, \mathtt{my})) \vee$
$(move_1(\ell_6, \ell_7) \wedge \mathtt{my} = 0 \wedge \mathtt{my}' = 1 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{mx}, \mathtt{a})) \vee$
$(move_1(\ell_7, \ell_8) \wedge \mathtt{y}' = \mathtt{y} + \mathtt{a} \wedge skip(\mathtt{x}, \mathtt{mx}, \mathtt{my}, \mathtt{a})) \vee$
$(move_1(\ell_8, \ell_9) \wedge \mathtt{my}' = 0 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{mx}, \mathtt{a})) \vee$
$(move_1(\ell_9, \ell_{10}) \wedge \mathtt{x}' = 2 * \mathtt{x} + \mathtt{a} \wedge skip(\mathtt{y}, \mathtt{mx}, \mathtt{my}, \mathtt{a})) \vee$
$(move_1(\ell_{10}, \ell_{11}) \wedge \mathtt{mx}' = 0 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{my}, \mathtt{a}))$

$next_2(V, V') =$
$(move_2(\ell_0, \ell_1) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{my})) \vee$
$(move_2(\ell_1, \ell_2) \wedge \mathtt{x}' = \mathtt{x} + 2 \wedge skip(\mathtt{y}, \mathtt{mx}, \mathtt{my})) \vee$
$(move_2(\ell_2, \ell_3) \wedge \mathtt{mx}' = 0 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{my}))$

$next_3(V, V') =$
$(move_3(\ell_0, \ell_1) \wedge \mathtt{my} = 0 \wedge \mathtt{my}' = 1 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{mx})) \vee$
$(move_3(\ell_1, \ell_2) \wedge \mathtt{y}' = \mathtt{y} + 2 \wedge skip(\mathtt{x}, \mathtt{mx}, \mathtt{my})) \vee$
$(move_3(\ell_2, \ell_3) \wedge \mathtt{my}' = 0 \wedge skip(\mathtt{x}, \mathtt{y}, \mathtt{mx}))$

$error(V) = (\mathtt{x} = 11 \wedge \mathtt{pc}_1 = \ell_{11} \wedge \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 = \ell_3)$

Fig. 1. Program P1-1 and its representation as a transition system. The notation $skip(v_1, \ldots, v_k)$ abbreviates the constraint $(v_1' = v_1 \wedge \cdots \wedge v_k' = v_k)$, while $move_i(\ell_j, \ell_k)$ stands for $(pc_i = \ell_j \wedge pc_i' = \ell_k)$.

II. Illustration

Our proposed method consists of two steps, inference of reducible blocks and compositional verification with summarization of the reducible blocks.

We illustrate our verification approach with a multi-threaded program that uses locks $(\mathtt{mx}, \mathtt{my})$ to protect accesses to the shared variables $(\mathtt{x}, \mathtt{y})$. See Figure 1 for the program P1-1. The program state is given by a valuation of program variables $V = (V_G, V_1, V_2, V_3)$, where V_G are global (shared) variables and V_i are thread-local variables for a thread $i \in \{1, 2, 3\}$. Each thread has a thread-local program counter variable $\mathtt{pc}_i \in V_i$ that holds location values ℓ_p for a program line labeled p. We denote by PC_i the set of locations from thread i, i.e., $PC_1 = \{\ell_0, \ldots, \ell_{11}\}$, $PC_2 = \{\ell_0, \ldots, \ell_3\}$ and $PC_3 = \{\ell_0, \ldots, \ell_3\}$. The assertion $init(V)$ gives the initial states of the program. We model the effect of program statements using a thread transition relation $next_i(V, V')$ corresponding to thread i. The primed version V' of the program variables are the next state valuations of V.

Our example uses a thread-synchronization model based on locks, acquire and release statements. We assume that a lock is initially not taken, e.g., $\mathtt{mx} = 0$ and $\mathtt{my} = 0$, and that the `acquire(mx)` statement waits until the lock is released $(\mathtt{mx} = 0)$ and then assigns the value 1 to mx. The release statement sets the lock value back to 0.

Reducible block boundaries The objective of the inference of reducible blocks is to minimize the number of explored interleavings during verification. For this illustration, we only show how reducible blocks are encoded in our approach. (See Section V for a formal description of a constraint based method for identifying commutativity of program statements and its application on the P1-1 example.)

Since global variables are consistently accessed while holding a corresponding lock, the statements between acquire and release for both `thread-2` and `thread-3` correspond to a reducible block. For `thread-1`, our inference obtains two reducible blocks, the first one from location ℓ_0 to ℓ_5 and the second one from ℓ_6 to ℓ_{10}. Informally, we shall refer to the four reducible blocks with labels (a), (b), (c) and (d).

(a) `thread-1` $\{\ell_0 - \ell_5\}$ (b) `thread-1` $\{\ell_6 - \ell_{10}\}$
(c) `thread-2` $\{\ell_0 - \ell_2\}$ (d) `thread-3` $\{\ell_0 - \ell_2\}$

Formally, the result of reduction is encoded using a partitioning of the transition relation of each thread into four categories: $next\_out\_out_i(V, V')$ describes transitions of thread i having both $\mathtt{pc}_i$ and $\mathtt{pc}_i'$ set to locations outside reducible blocks, $next\_in\_in_i(V, V')$ and $next\_out\_in_i(V, V')$ describe transitions with target location $\mathtt{pc}_i'$ inside a reducible block, while $next\_in\_out_i(V, V')$ describes transitions having $\mathtt{pc}_i$ set to a location inside a reducible block and target $\mathtt{pc}_i'$ outside a reducible block. We also make sure that transitions that target error locations are not part of reducible blocks.

Compositional proof rule with reduction The crux of our verification approach is a proof rule that uses both reduction and compositional reasoning. The proof rule lists conditions over three kinds of auxiliary assertions (or program invariants):

$IR_i(V)$ describes reachable states outside reducible blocks that should be accounted for interference by different threads; $LStep_i(V_G, V_i, V'_G, V'_i)$ is a binary relation representing steps inside the same reducible block that are visible only to thread i; $IStep_i(V, V')$ represents steps of thread i that are visible to other threads including steps outside reducible blocks and summaries of reducible blocks.

Let us consider an interleaving of program statements that starts with those statements from the reducible block (c). Verification based on our proof rule continues exploring non-deterministically reducible blocks from either `thread-1` or `thread-3`, i.e., $c{-}a{-}b{-}d$ or $c{-}a{-}d{-}b$ or $c{-}d{-}a{-}b$. Few interleavings are effectively explored due to the coarse-grained nature of reducible blocks. Overall, the following list contains possible block-interleavings that are explored for P1-1.

(I1)	$a-b-c-d$	(I7)	$c-a-b-d$
(I2)	$a-b-d-c$	(I8)	$c-a-d-b$
(I3)	$a-c-b-d$	(I9)	$c-d-a-b$
(I4)	$a-c-d-b$	(I10)	$d-a-b-c$
(I5)	$a-d-b-c$	(I11)	$d-a-c-b$
(I6)	$a-d-c-b$	(I12)	$d-c-a-b$

The effect of all these interleavings is captured by the auxiliary assertions from our proof rule.

For illustration, we aim to prove that the value of the variable `x` is not equal to 11 at the end location. (The variable `x` could have value 11 at the end of the program only following the interleaving $a-c-b-d$. However, as the reader may observe, this interleaving corresponds to an infeasible execution.) For safety, we require that the auxiliary assertions corresponding to the reachable states do not intersect error states. The proof rule has premises over the auxiliary assertions that are expressed as universally quantified Horn clauses. (See Section IV for the formal details.) We compute solutions for the auxiliary assertions using a Horn solver based on abstraction refinement and interpolation over the linear arithmetic domain [14].

For the illustration example, the solution for the reachable state assertion is computed as follows.

$$
\begin{array}{l}
\mathtt{pc}_1 = \ell_0 \wedge \mathtt{mx} = 0 \wedge \\
\quad (\mathtt{pc}_2 = \ell_0 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} = 2 \vee \\
\quad\ \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge 4 \leq \mathtt{x} \leq 7) \vee \\
\mathtt{pc}_1 = \ell_6 \wedge \mathtt{mx} = 1 \wedge \\
\quad (\mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} = 2 \wedge 2\mathtt{x} + \mathtt{a} = 7 \vee \\
\quad\ \mathtt{pc}_2 \in \{\ell_0, \ell_3\} \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge 4 \leq \mathtt{x} \leq 7 \wedge 2\mathtt{x}+\mathtt{a} \geq 13) \vee \\
\mathtt{pc}_1 = \ell_{11} \wedge \mathtt{mx} = 0 \wedge \\
\quad (\mathtt{pc}_2 = \ell_0 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \leq 7 \vee \\
\quad\ \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \leq 9 \vee \\
\quad\ \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 \in \{\ell_0, \ell_3\} \wedge \mathtt{x} \geq 13 \wedge 2\mathtt{x} + \mathtt{a} \geq 13)
\end{array}
$$

All the states have the location of `thread-1`, $\mathtt{pc}_1 \in \{\ell_0, \ell_6, \ell_{11}\}$. The lock `mx` is held only at ℓ_6, otherwise it is available. The different cases for each program location result from varied interleavings of `thread-1` and `thread-2`, since statements of `thread-3` have no influence on the value of `x`. At states with $\mathtt{pc}_1 = \ell_{11}$, we observe three possibilities: `thread-2` has not yet started ($\mathtt{x} \leq 7$), `thread-2` may have been executed after `thread-1` ($\mathtt{x} \leq 9$), or `thread-2` may have been executed before `thread-1` ($\mathtt{x} \geq 13$). Note that our method over-approximates the set of reachable states, e.g., constraints on value of `y` are not present in the above solution.

III. Preliminaries

Multi-threaded programs A multi-threaded program P consists of $N \geq 1$ threads. We assume that the program variables $V = (V_G, V_1, \ldots, V_N)$ are partitioned into global variables V_G shared by all threads and local variables $V_1, \ldots, V_N$, which are only accessible by the respective threads.

The set of global states G consists of the valuations of global variables, and the sets of local states $L_1, \ldots, L_N$ consist of the valuations of the local variables of respective threads. A program state is a valuation of the global variables and the local variables of all threads. We represent sets of program states using assertions over program variables. Binary relations between sets of program states are represented using assertions over unprimed and primed variables. The set of initial program states is denoted symbolically by $init(V)$. For each thread i we have a finite set of transitions. Each transition is a binary relation between sets of program states. Furthermore, each transition can only change the values of the global variables and the local variables of the thread i (local variables of other threads do not change). This fact is captured in constraint form using the abbreviation $next_i^{=} := \bigwedge_{j \in 1..N \setminus \{i\}} V'_j = V_j$. We write $next_i(V, V')$ for the union of the transitions of the thread i . The transition relation of the program is $next(V, V') = next_1(V, V') \wedge next_1^{=}(V, V') \vee \cdots \vee next_N(V, V') \wedge next_N^{=}(V, V')$. In the subsequent sections, we abbreviate $next_i(V, V') \wedge next_i^{=}(V, V')$ to $next_i(V, V')$.

We distinguish two special types of variables, program counter variables and lock variables. Firstly, each thread has a program counter pc_i that is a local variable with values in the set PC_i. As a convention, we use labels $\ell_0, \ell_1, ..$ to denote some elements from the previous set. Secondly, some global variables are used for thread synchronization via acquire (acq) and release (rel) primitives. The set of lock variables is denoted by $Locks$, we have $Locks \subseteq V_G$ and we use m, mx, my to denote some elements from the set of locks.

Computations Let $\models$ denote the satisfaction relation between (pairs) of states and assertions over program variables (and their primed versions). A computation of P is a sequence of program states $s_1, s_2, \ldots$ such that s_1 is an initial state, i.e., $s_1 \models init$, and each pair of consecutive states s_i and s_{i+1} in the sequence is connected by a transition ρ of some program thread, i.e., $(s_i, s_{i+1}) \models \rho$. A path is a sequence of transitions.

A program state is reachable if it appears in some computation. Let φ_{reach} be the symbolic representation of the set of all reachable states. The set of error states of a program is denoted using $error(V)$. The program is safe if none of its error states is reachable, i.e., $\varphi_{reach}(V) \wedge error(V) \rightarrow false$. The program is *terminating* if it does not have any infinite computations.

Constraints and queries Let $\mathcal{T}$ be a first-order theory in a given signature and $\models_{\mathcal{T}}$ be the entailment relation with respect to $\mathcal{T}$. We refer to formulas in the given signature as constraints, and let $c(v)$ denote a constraint over the variables v. For example, let x, y, and z be variables. Then, $v = (x, y)$ and $w = (y, z)$ are tuples of variables. $x \leq 2$, $y \leq 1 \wedge x - y \leq 0$ are example constraints in the theory $\mathcal{T}$ of linear inequalities over rationals/reals. The entailment $y \leq 1 \wedge x - y \leq 0 \models_{\mathcal{T}} x \leq 2$ is valid.

ISBN: 978-0-9835678-4-4.

For assertions IR_i, $LStep_i$ and $IStep_i$,

$$
\begin{array}{lll}
\text{(S1)} & \mathit{init}(V) & \rightarrow \mathit{IR}_i(V) \\
\text{(S2)} & \mathit{IR}_i(V) \wedge \mathit{next\_out\_in}_i(V, V') & \rightarrow \mathit{LStep}_i(V_G, V_i, V'_G, V'_i) \\
\text{(S3)} & \mathit{LStep}_i(V_G, V_i, V'_G, V'_i) \wedge \mathit{next\_in\_in}_i(V', V'') & \rightarrow \mathit{LStep}_i(V_G, V_i, V''_G, V''_i) \\
\text{(S4)} & \mathit{IR}_i(V) \wedge \mathit{LStep}_i(V_G, V_i, V'_G, V'_i) \wedge \mathit{next}_i^{=}(V, V') \wedge \mathit{next\_in\_out}_i(V', V'') & \rightarrow \mathit{IStep}_i(V, V'') \wedge \mathit{IR}_i(V'') \\
\text{(S5)} & \mathit{IR}_i(V) \wedge \mathit{next\_out\_out}_i(V, V') & \rightarrow \mathit{IStep}_i(V, V') \wedge \mathit{IR}_i(V') \\
\text{(S6)} & \mathit{IR}_i(V) \wedge (\bigvee_{j \in 1..N \setminus \{i\}} \mathit{IStep}_j(V, V')) & \rightarrow \mathit{IR}_i(V') \\
\text{(S7)} & (\bigwedge_{i=1}^{N} \mathit{IR}_i(V)) \wedge \mathit{error}(V) & \rightarrow \mathit{false}
\end{array}
$$

multi-threaded program P is safe

Fig. 2. Proof rule RULESAFETY.

We assume a set of uninterpreted predicate symbols $\mathcal{Q}$ that we refer to as query symbols. The arity of a query symbol is assumed to be encoded in its name. We write q to denote a query symbol. Given q of a non-zero arity n and a tuple of variables v of length n, we define $q(v)$ to be a query. For example, let $\mathcal{Q} = \{r, s\}$ be query symbols of arity one and two, respectively. Then, $r(x)$ and $s(x, y)$ are queries.

Horn-like clauses Let $h(v)$ range over queries and constraints with variables in v. We define a Horn-like clause to be an implication $c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow h(v)$. The left-hand side of the implication is called the body and the right-hand side is called the head. To support efficient verification, our Horn-like clauses slightly deviate from the standard notion of Horn clauses since constraints occurring in our clauses can contain disjunctions and conjunctions.

Solving Horn-like clauses We use a solver for Horn clauses over a first-order theory $\mathcal{T}$ that is invoked as follows.

$$\Sigma := \text{HSF}(\mathit{HC}, \mathcal{Q}, \mathit{Preds})$$

The solver takes as input a set of clauses HC over queries $\mathcal{Q}$ with optional predicates $Preds$. The function $Preds$ assigns a finite set of predicates to each query symbol q from $\mathcal{Q}$ and defines the abstract domain of a data-flow analysis or predicate abstraction. The solver returns a solution function Σ that maps each query from $\mathcal{Q}$ to a constraint from $\mathcal{T}$.

IV. PROOF RULES

In this section we present proof rules that combine reduction and compositional reasoning.

A. Proof rule for safety

See Figure 2 for our proof rule RULESAFETY that lists conditions for program safety over the following assertions.

- $IR_i(V)$: interfering state assertions that represent state reachability information outside reducible blocks for thread $i \in 1..N$.
- $LStep_i(V_G, V_i, V'_G, V'_i)$: non-interfering step assertions that represent steps of thread i that are only locally-visible for thread $i \in 1..N$.
- $IStep_i(V, V')$: interfering step assertions that represent steps of thread i that are visible to other threads (interfering steps) for thread $i \in 1..N$.

The clauses (S1) to (S6) are replicated for each thread i. The clause (S1) considers that initial states are reachable states. The clauses (S2) and (S3) do thread-modular reasoning inside reducible blocks - (S2) initiates relations with target locations inside reducible blocks and (S3) transitively extends these relations. The clause (S4) makes the effect of a reducible block visible to other threads, as well as in the interfering reachable states. The clauses (S5) and (S6) perform compositional reasoning outside reducible blocks by using single transitions and reducible block relations, respectively. The last clause (S7) checks that states reachable outside reducible blocks do not intersect the error states.

Theorem 1. *The proof rule* RULESAFETY *is sound, i.e., if an error state is reachable the constraint system consisting of clauses (S1) to (S7) has no solution.*

A correctness argument of the proof rule is omitted for space constraints. (A soundness proof for a rule based on reduction and compositional reasoning is included in the thesis of one of the authors [15, (Section 3.5)].)

Example 1. The first clause from the proof rule states that all initial states are included in the $IR_i(V)$ assertions. For the example from Section II a solution of the reachable-states assertion will include at least the initial states:

$$
\begin{array}{l}
(\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_0 \wedge \mathtt{pc}_3 = \ell_0 \wedge \\
\quad \mathtt{x} = 2 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0)
\end{array}
$$

Clause (S2) initiates a binary relation $LStep_i$ for a thread i whenever a transition $\mathit{next\_out\_in}_i(V, V')$ targeting a location from a reducible block is enabled. Once inside a reducible block, the clause (S3) uses relational composition to include relations in $LStep_i$ as long as further transitions $\mathit{next\_in\_in}_i(V, V')$ are enabled. We illustrate the application of these clauses using the transitions corresponding to `thread-2` that start from the previously computed initial states.

$$
\begin{array}{l}
\mathit{move}_2(\ell_0, \ell_1) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge \mathit{skip}(\mathtt{x}, \mathtt{y}, \mathtt{my}) \vee \\
\mathit{move}_2(\ell_0, \ell_2) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 1 \wedge \mathtt{x}' = \mathtt{x} + 2 \wedge \mathit{skip}(\mathtt{y}, \mathtt{my})
\end{array}
$$

For assertions IR_i, $LStep_i$ and $IStep_i$ satisfying (S1), ...,(S6)
and assertions $LRound_i$, $IRound$,

$$\begin{array}{ll}
\text{(T1)} & (\bigwedge_{i=1}^{N} IR_i(V)) \wedge LStep_i(V_G, V_i, V_G', V_i') \wedge next_in_in_i(V', V'') \rightarrow LRound_i(V_G', V_i', V_G'', V_i'') \\
\text{(T2)} & well\text{-}founded(LRound_i) \\
\text{(T3)} & (\bigwedge_{i=1}^{N} IR_i(V)) \wedge (\bigvee_{j=1}^{N} IStep_j(V, V')) \rightarrow IRound(V, V') \\
\text{(T4)} & well\text{-}founded(IRound)
\end{array}$$

multi-threaded program P terminates

Fig. 3. Proof rule RULETERMINATION.

Clause (S4) generates a summary relation for the reducible block of `thread-2` from the previous relation and the assertion $next\_in\_out_2(V, V')$:

$$move_2(\ell_0, \ell_3) \wedge \mathtt{mx} = 0 \wedge \mathtt{mx}' = 0 \wedge \mathtt{x}' = \mathtt{x} + 2 \ \wedge skip(\mathtt{y}, \mathtt{my})$$

Besides clause (S1), the clauses (S4) and (S5) generate reachable states $IR_i(V)$ by applying enabled reducible block relations or transitions outside reducible blocks, respectively. For our example, the following formula represents additional reachable states generated from these clauses.

$$\begin{array}{l}
(\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_0 \wedge \mathtt{pc}_3 = \ell_0 \ \wedge \\
\quad \mathtt{x} = 2 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0) \ \vee \\
(\mathtt{pc}_1 = \ell_0 \wedge \mathtt{pc}_2 = \ell_3 \wedge \mathtt{pc}_3 = \ell_0 \ \wedge \\
\quad \mathtt{x} = 4 \wedge \mathtt{y} = 2 \wedge \mathtt{mx} = 0 \wedge \mathtt{my} = 0)
\end{array}$$

B. Proof rule for termination

See Figure 3 for our proof rule RULETERMINATION that lists conditions to ensure that a program is terminating. The conditions are over assertions IR_i, $LStep_i$, $IStep_i$ that satisfy the clauses from the rule RULESAFETY and the following additional assertions:

- $LRound_i$: binary relation assertions that represent thread-modular transition relations inside reducible blocks for $i \in 1..N$.
- $IRound$: binary relation assertion that represents transition relations outside of reducible blocks together with summary relations of reducible blocks.

For each thread i the clause (T1) together with clause (T2) guarantees that there is no infinite computation executing within some reducible block. Clause (T3) together with clause (T4) guarantees that there is no infinite computation that keeps alternating between reducible blocks of the program infinitely often.

Theorem 2. *The proof rule* RULETERMINATION *is sound, i.e., the constraint system consisting of clauses (S1)..(S6) and (T1)..(T4) has a solution only if the program is terminating.*

The premises of RULETERMINATION can be solved using the Horn solver HSF [14], since the premises can be represented as Horn clauses with disjunctive well-foundedness constraints. We write $well\text{-}founded(\varphi(v, v'))$ if $\varphi(v, v')$ is a well-founded relation, i.e., there is no infinite sequence $s_1, s_2, \ldots$ such that $\varphi(s_i, s_{i+1})$ for all $i > 1$. A relation $\varphi(v, v')$ is disjunctively well-founded if it is included in a finite union of well-founded relations, i.e., there exist well-founded $\varphi_1(v, v'), \ldots, \varphi_n(v, v')$ such that $\varphi(v, v') \rightarrow \varphi_1(v, v') \vee \cdots \vee \varphi_n(v, v')$ is a valid implication.

Example 2. We extend `thread-1` from Figure 1 with a loop that spans over newly inserted locations ℓ_{1b} and ℓ_{8b}.

```
// Thread-1
 ...
 1:  a = x;
 1b: while a<=4
 ...
 8:  release(my);
 8b: endwhile
 9:  x = 2*x+a;
 ...
```

Since this change does not introduce any additional non-mover transitions, the reducible block boundaries of `thread-1` remain the same, i.e., $\{\ell_0 - \ell_5\}$ and $\{\ell_6 - \ell_{10}\}$.

The check corresponding to clause (T2) succeeds immediately, since the example does not contain looping executions in a reducible block. For (T3), consider the following formula that is computed by the Horn solver for the body of the clause.

$$\begin{array}{l}
(\bigwedge_{i=1}^{N} IR_i(V)) \wedge \Big(\big(move_1(\ell_0, \ell_6) \wedge \mathtt{a} \leq 4 \wedge ..\big) \ \vee \\
\quad \big(move_1(\ell_6, \ell_6) \wedge \mathtt{a} \leq 4 \wedge \mathtt{a}' = \mathtt{a} + 1 \wedge ..\big) \ \vee \\
\quad \big(move_1(\ell_6, \ell_{11}) \wedge \mathtt{a} > 4 \wedge ..\big) \ \vee \\
\quad \big(move_2(\ell_0, \ell_3) \wedge ..\big) \ \vee \\
\quad \big(move_3(\ell_0, \ell_3) \wedge ..\big) \Big)
\end{array}$$

The only disjunct that could potentially permit infinite state sequences corresponds to $move_1(\ell_6, \ell_6)$. The HSF solver concludes that this relation is well-founded, since variable `a` is incremented and has an upper bound, and thus the example program is proven terminating.

V. INFERENCE OF REDUCIBLE BLOCKS

This section depicts the computation steps for obtaining reducible block boundaries. We consistently use a constraint-based approach to solve the data-flow problems for every step. Our formalization is based on the theory of reduction [6] and follows the approach used to infer transactions for finite-state model checking [9]. We illustrate our method using the program from Figure 1.

A. Locks-held and mover information

Reducible block inference requires for each reachable program transition specific mover information which highly depends on the held locks. We use data-flow analysis to compute $lh_i(\ell)$, an approximation of the set of locks held by a thread $i \in 1..N$ at location $\ell \in PC_i$. The following set of Horn clauses over queries $\mathcal{Q}_1 := \{LR_1(V), \ldots LR_N(V)\}$ allows us to obtain reachable states of thread i without considering any thread context switches.

$$\begin{aligned} HC_1 := \{\ & init(V) \rightarrow LR_i(V), \\ & LR_i(V) \wedge next_i(V, V') \rightarrow LR_i(V') \mid i \in 1..N\} \end{aligned}$$

The abstract domain of the static analysis is defined by a predicate function. It is initialized with predicates over program counter and lock variables as follows.

$$\begin{aligned} Preds_1(LR_i(V)) := \ & \{pc_i = \ell_i \mid \ell_i \in PC_i\} \cup \\ & \{m = 0, m = 1 \mid m \in Locks\} \end{aligned}$$

We invoke the HSF solver: $\Sigma_1 := \text{HSF}(HC_1, \mathcal{Q}_1, Preds_1)$. Note that without a clause involving error states, the solver computes only one over-approximation of the reachable states, i.e. no abstraction refinement is performed in this phase. $lh_i(\ell)$ contains the set of held locks for location ℓ by utilizing Σ_1.

$$lh_i(\ell) := \{m \in Locks \mid \forall V{:}\Sigma_1(LR_i(V)) \wedge pc_i = \ell \rightarrow m = 1\}$$

Example 3. The solution corresponding to the first thread from Figure 1, $\Sigma_1(LR_1(V))$, follows.

$$\begin{aligned} & (pc_1 \in \{\ell_0, \ell_{11}\} \wedge mx = 0 \wedge my = 0) \vee \\ & (pc_1 \in \{\ell_1, \ell_2, \ell_5, \ell_6, \ell_9, \ell_{10}\} \wedge mx = 1 \wedge my = 0) \vee \\ & (pc_1 \in \{\ell_3, \ell_4, \ell_7, \ell_8\} \wedge mx = 1 \wedge my = 1) \end{aligned}$$

The locks-held information derived at location ℓ_3 is $lh_1(\ell_3) := \{mx, my\}$.

We represent transition-mover information using four boolean functions defined over pairs of program locations: $rm_i(pc_i, pc_i')$, $lm_i(pc_i, pc_i')$, $nm_i(pc_i, pc_i')$, $bm_i(pc_i, pc_i')$. Following the theory of reduction [6], an acquire transition is a right-mover (i.e., it commutes to the right with every transition from other threads) and a release transition is a left-mover (i.e., it commutes to the left with every transition from other threads). A transition $\rho_i(pc_i, pc_i')$ is a non-mover if there exists a transition from another thread $\rho_j(pc_j, pc_j')$ that accesses some common global variable (at least one thread performing a write access) and the intersection of the sets of locks held by thread i at pc_i and those held by thread j at pc_j is empty. Transitions that are neither left-movers, right-movers nor non-movers are both-movers.

Example 4. For the first thread we obtain:

$$\begin{aligned} rm_1 &:= \{(\ell_0, \ell_1), (\ell_2, \ell_3), (\ell_6, \ell_7)\} \\ lm_1 &:= \{(\ell_4, \ell_5), (\ell_8, \ell_9), (\ell_{10}, \ell_{11})\} \\ nm_1 &:= \emptyset \\ bm_1 &:= \{(\ell_1, \ell_2), (\ell_3, \ell_4), (\ell_5, \ell_6), (\ell_7, \ell_8), (\ell_9, \ell_{10})\} \end{aligned}$$

B. In-Out information

Let $n, m \in \mathbb{Z}^+$, $i \in 1..n$, and $j \in 1..m$. A reducible block is a non-empty sequence of transition relations $a_1, \ldots, a_n, [c], b_1, \ldots, b_m$ where each a_i (b_j) is a right-mover (left-mover) and c is an optional non-mover. We use the transition-mover information from Section V-A to group program locations into two phases; a *pre-commit-phase* and a *post-commit-phase*. The former phase contains target locations of right-mover (a_i) or initial locations. The latter phase contains target locations of all other transitions (c, b_j).

We utilize Horn clauses over the following set of queries: $\mathcal{Q}_2 := \{Ph_1(V, p), \ldots, Ph_N(V, p)\}$ representing reachable state queries extended by a boolean phase variable p that indicates either the *pre-commit-phase* (p has value 1) or the *post-commit-phase* (p has value 0). The set HC_2 contains the following clauses replicated for $i \in 1..N$.

$$\begin{aligned} & init(V) \wedge p = 1 && \rightarrow Ph_i(V, p) \\ & Ph_i(V, p) \wedge next_i(V, V') \wedge rm_i(pc_i, pc_i') \wedge p' = 1 && \rightarrow Ph_i(V', p') \\ & Ph_i(V, p) \wedge next_i(V, V') \wedge \\ & \quad (lm_i(pc_i, pc_i') \vee nm_i(pc_i, pc_i')) \wedge p' = 0 && \rightarrow Ph_i(V', p') \\ & Ph_i(V, p) \wedge next_i(V, V') \wedge bm_i(pc_i, pc_i') \wedge p' = p && \rightarrow Ph_i(V', p') \end{aligned}$$

To define the abstract domain of the data-flow analysis, we initialize the predicate function with predicates over program counter and phase variables.

$$Preds_2(Ph_i(V, p)) := \{pc_i = \ell_i \mid \ell_i \in PC_i\} \cup \{p{=}0, p{=}1\}$$

We invoke the HSF solver: $\Sigma_2 := \text{HSF}(HC_2, \mathcal{Q}_2, Preds_2)$. Reducible block information is extracted from the solution Σ_2 and represented using boolean functions defined over program locations. $In_i(pc_i)$ holds when pc_i is a location inside a reducible block, while $Out_i(pc_i)$ holds when pc_i is a location outside any reducible block. A location is inside a reducible block if it is contained in the *pre-commit-phase* or if every enabled transition left-commutes with transitions from other threads. Otherwise, a location is outside any reducible block.

$$\begin{aligned} In_i(pc_i) \ & := \Sigma_2(Ph_i(V, p)) \wedge \neg init(V) \wedge (p = 1 \vee p = 0 \wedge \\ & \quad \forall pc_i' : lm_i(pc_i, pc_i') \vee bm_i(pc_i, pc_i')) \\ Out_i(pc_i) \ & := \neg In_i(pc_i) \end{aligned}$$

Example 5. The solution corresponding to the first thread follows.

$$\begin{aligned} \Sigma_2(Ph_1(V, p)) := (\ & pc_1 \in \{\ell_0, \ell_1, \ell_2, \ell_3, \ell_4\} \wedge p = 1 \vee \\ & pc_1 \in \{\ell_5\} \wedge p = 0 \vee \\ & pc_1 \in \{\ell_6, \ell_7, \ell_8\} \wedge p = 1 \vee \\ & pc_1 \in \{\ell_9, \ell_{10}, \ell_{11}\} \wedge p = 0) \end{aligned}$$

We obtain the following results for the first thread: $Out_1 := \{\ell_0, \ell_6, \ell_{11}\}$ and $In_1 := PC_1 \setminus Out_1$. The results for the second and third thread are computed similarly: $Out_2 := \{\ell_0, \ell_3\}, In_2 = \{\ell_1, \ell_2\}$ and $Out_3 := \{\ell_0, \ell_3\}, In_3 = \{\ell_1, \ell_2\}$.

Given the in-out information, we partition the transition relation of a thread depending on whether the target of a transition is a state in/outside a reducible block. We also make sure that transitions that target error locations are not part of reducible blocks. We obtain the following four relations corresponding to $next\_in\_in_i(V, V')$, $next\_out\_in_i(V, V')$, $next\_in\_out_i(V, V')$ and respectively to $next\_out\_out_i(V, V')$.

$$\begin{aligned} & next_i(V, V') \wedge In_i(pc_i) \wedge In_i(pc_i') \wedge \neg error(V') \\ & next_i(V, V') \wedge Out_i(pc_i) \wedge In_i(pc_i') \wedge \neg error(V') \\ & next_i(V, V') \wedge In_i(pc_i) \wedge (Out_i(pc_i') \vee error(V')) \\ & next_i(V, V') \wedge Out_i(pc_i) \wedge (Out_i(pc_i') \vee error(V')) \end{aligned}$$

TABLE I. RESULTS FOR VERIFICATION OF SAFETY PROPERTIES. A ✓-MARK (×-MARK) INDICATES A SAFE (UNSAFE) PROGRAM. EXPERIMENTS WERE RUN ON AN INTEL XEON MACHINE, CLOCKED AT 3.47GHZ WITH 8 GB RAM. A T/O-MARK REPRESENTS A TIME-OUT AFTER 5400S.

Program	LOC	Threads	Safe	Impara	Threader	Comp	RedComp
P1-1	48	3	✓	**1s**	6s	7s	2s
P1-5	64	3	✓	110s	281s	101s	**32s**
P1-10	84	3	✓	T/O	T/O	840s	**64s**
P1-50	244	3	✓	T/O	T/O	T/O	**2400s**
P2-5	65	3	✓	**83s**	617s	270s	140s
P2-10	85	3	✓	T/O	T/O	1020s	**220s**
P2-50	245	3	✓	T/O	T/O	T/O	**3778s**
stack-safe-5	50	3	✓	115s	**5s**	96s	17s
stack-safe-10	50	3	✓	635s	127s	224s	**75s**
stack-unsafe-5	48	3	×	2s	**1s**	9s	2s
stack-unsafe-10	48	3	×	62s	**2s**	9s	3s
pbzip2-safe	283	4	✓	T/O	T/O	T/O	**840s**
twostage-3-unsafe	129	4	×	T/O	843s	T/O	**17s**

VI. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

The implementation of our approach consists of three conceptual modules. The first module is a frontend implemented in the OCaml language that relies on the CIL library. It translates C programs to corresponding transition systems in Horn clause form. Our frontend relies on a number of static analyses: thread-scope inference for dynamic thread creation, a pointer analysis that is context-sensitive and malloc-sensitive, optional array expansion for bounded arrays and restricted quantified invariants for unbounded arrays.

A second module that infers reducible block boundaries following the approach from Section V can be automated using HSF, but is not yet integrated in our implementation. The lock analysis can be realised through solving the clauses HC1 from Section 5.1, while identification of left/right-movers reduces to solving the clauses HC2 from Section 5.2.

The third module is a model checker implemented using the HSF approach [14]. This module is given as input a proof rule written as Horn clauses and the program generated by our frontend with thread transition relations partitioned as $next\_in\_in$, $next\_in\_out$, $next\_out\_in$ and $next\_out\_out$.

A. Evaluation

In this section, we give details on an experimental evaluation of our approach. We compare results from our implementation with two state-of-the-art verifiers: Threader [16], the winner in the Concurrency category of SV-COMP 2013 and Impara [5], a verifier that combines partial-order-reduction and interpolation [17]. Binaries and test programs used for evaluation are made publicly available [18].

In general, our method benefits from the datarace-free nature of statements to infer coarse-grained reducible blocks. For evaluation purposes, we test how our verifier works on programs with race conditions on shared variables. Consider P2-1, a variation of P1-1 from Figure 1 such that `thread-3` has an additional statement that accesses the shared variable `y` without holding the lock `my`. For this modified program, our reducible block inference computes more reducible blocks for `thread-1` and `thread-3` than it is the case for the original program P1-1. However, the reduction phase still significantly reduces the number of interleavings to be explored.

See Table I for verification results of safety properties. We report on variations of four programs. P1-1 and P2-1 were described in the previous sections of the paper, stack-safe-5 is part of SV-COMP 2013 and is challenging to the partial-order reduction method implemented in Impara [5] and stack-unsafe-5 is the modified stack example that does not satisfy its safety assertion. As variations, P1-x, P2-x have "x" statements in each of their reducible blocks. For stack-safe-x and stack-unsafe-x, we vary the number of elements stored in the stack. Lastly, we include two benchmarks that are challenging to Impara and Threader, twostage-3-unsafe from SV-COMP and pbzip2-safe, a multi-threaded implementation of a compression algorithm.

For each test program from Table I, we report the number of lines of C code in Column 2, the expected verification result in Column 4 and statistics on four verification methods. Column 5 presents results from Impara [5], while Column 6 presents timings from the best performing compositional proof rule implemented in Threader. Column 7 presents results from our implementation based on a rule that uses compositional reasoning but not reduction. Column 8 presents timings for our new verification method (REDCOMP stands for the Reduction-Compositional verification).

For the same implementation, we observe that reduction improves the performance of a compositional reasoning verifier, i.e., REDCOMP in Column 7 versus Comp in Column 6. When comparing our synthesis-driven implementation Comp with THREADER, a verifier optimized for the same proof rule, we observe some overhead for test programs that are less favorable for reduction, i.e., P1-1, stack-safe-5 and stack-unsafe-5. However, for the variations of these programs that are more favorable for reduction, we observe reduction in verification time for our proposed method REDCOMP.

See Table II for results on verification of termination properties. The program fig2-tacas12 has a complex termination proof based on disjunctive well-founded transition invariant [19]. sync01-safe is a benchmark from SV-COMP 2014 that is marked as safe for assertion violations and suffers from a non-termination bug. One thread may block waiting for a signal on a condition variable, a bug that is uncovered using REDCOMP. Finally, we include a C program modeling the dining philosophers problem.

Due to the not-yet integrated block inference, we present in this section only a limited experimental evaluation on selected examples that are challenging for Impara and Threader. In principle our current approach (reduction + compositional reasoning) subsumes the compositional algorithms from Threader. For the most imprecise inference of reducible blocks, i.e., with $In_i = \emptyset$ and $Out_i = PC_i$, the proof rule from Figure 2

TABLE II. RESULTS FOR VERIFICATION OF TERMINATION PROPERTIES.

Program	LOC	Terminates	Comp	RedComp
fig2-tacas12	24	✓	**2s**	3s
sync01-safe-fixed	62	✓	308s	**4s**
dining-philo	108	✓	T/O	**7s**

reduces immediately to the Owicki-Gries rule automated in Threader. (Threader already delivers conclusive results for most of the other "Concurrency" benchmarks from SV-COMP.)

VII. RELATED WORK

The reduction principle, as formulated by Lipton [6], has been used in program analysis for checking or inferring whether a method is atomic, i.e., whether the body of the method corresponds to a reducible block. These program analyses were formalized either using a type system [12], [20], or as a dynamic analysis [21]. Going one step further and using the result of atomicity analysis for verification has been proposed only in the context of finite-state verification algorithms [8], [9] where the algorithms that benefit from reduction are quite different than approaches like ours based on interpolation-based verification. Reduction can greatly simplify deductive verification of multi-threaded programs using proof assistants [7]. Our current work can be viewed as a step towards an integration of reduction in interpolation-based verification.

Apart from works based directly on Lipton's theory of reduction, there have been other verification methods aiming to avoid exploring interleavings that are equivalent.

One approach stems from compositional reasoning proof rules, i.e., the Owicki-Gries method [11] or rely-guarantee reasoning [22]. These compositional proof methods have been automated for verification of finite-state models [23] and infinite-state models [24] using counter-example guided abstraction refinement [25]. Since compositional reasoning avoids exploring many equivalent interleavings, Threader [16], an implementation of the previous algorithms, has been able to compete with success in the Concurrency category of the verification competition held at TACAS [26]. Our current work can be viewed as an extension of Threader's algorithms with a reduction-based static analysis that avoids exploring even more redundant interleavings.

Another approach to the state explosion problem is partial order reduction [1] that has been used for finite-state verification, e.g., [2]. Recent work shows that POR can also boost interpolation based verification [5], which makes it applicable for the verification of programs with infinite-state spaces. This approach has been implemented in a tool called Impara.

We emphasize the connection between procedure summarization [13] and our approach. Rather than summarizing procedures in sequential programs, our current work summarizes reducible blocks in the context of multi-threaded program verification. Our approach has been inspired by a work on summarization of concurrent programs [9], with the distinguishing feature that our work is applicable for infinite-state spaces. While procedure summarization allowed software analysis tools like SLAM and SATURN to perform composable analysis of large code bases, our work aims to use summarization of reducible blocks to allow verification to scale to large multi-threaded programs.

ACKNOWLEDGMENTS

We thank Klaus von Gleissenthall for comments and suggestions. This research was supported in part by the ERC project 308125.

REFERENCES

[1] P. Godefroid, "Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem," Ph.D. dissertation, University of Liege, Computer Science Department, 1994.

[2] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL*, 2005.

[3] F. Lerda, N. Sinha, and M. Theobald, "Symbolic model checking of software," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 3, pp. 480–498, 2003.

[4] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-order reduction in symbolic state space exploration," in *CAV*, 1997.

[5] B. Wachter, D. Kroening, and J. Ouaknine, "Verifying multi-threaded software with Impact," in *FMCAD*, 2013.

[6] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, pp. 717–721, 1975.

[7] T. Elmas, S. Qadeer, and S. Tasiran, "A calculus of atomic actions," in *POPL*, 2009.

[8] C. Flanagan and S. Qadeer, "Transactions for software model checking," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 3, pp. 518–539, 2003.

[9] S. Qadeer, S. K. Rajamani, and J. Rehof, "Summarizing procedures in concurrent programs," in *POPL*, 2004.

[10] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, "Zing: A model checker for concurrent software," in *CAV*, 2004.

[11] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Inf.*, vol. 6, 1976.

[12] C. Flanagan and S. Qadeer, "Types for atomicity," in *TLDI*, 2003.

[13] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL*, 1995.

[14] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*, 2012.

[15] A. Wilhelm, "Efficient verification of multi-threaded programs," Master's thesis, 2013, available from http://www.model.in.tum.de/~popeea/research/wilhelm.msc13.pdf/.

[16] A. Gupta, C. Popeea, and A. Rybalchenko, "Threader: A constraint-based verifier for multi-threaded programs," in *CAV*, 2011.

[17] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV*, 2006.

[18] C. Popeea, "Redcomp webpage," http://www.model.in.tum.de/~popeea/research/redcomp, accessed: 09-Feb-2014.

[19] C. Popeea and A. Rybalchenko, "Compositional termination proofs for multi-threaded programs," in *TACAS*, 2012.

[20] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *PLDI*, 2003.

[21] C. Flanagan and S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," in *POPL*, 2004.

[22] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, 1983.

[23] A. Cohen and K. S. Namjoshi, "Local proofs for global safety properties," in *CAV*, 2007.

[24] A. Gupta, C. Popeea, and A. Rybalchenko, "Predicate abstraction and refinement for verifying multi-threaded programs," in *POPL*, 2011.

[25] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, 2000.

[26] D. Beyer, "Second competition on software verification - (summary of SV-COMP 2013)," in *TACAS*, 2013.