Compositional Termination Proofs for Multi-threaded Programs

Corneliu Popeea and Andrey Rybalchenko

Technische Universität München

Abstract. Automated verification of multi-threaded programs is difficult. Direct treatment of all possible thread interleavings by reasoning about the program globally is a prohibitively expensive task, even for small programs. Rely-guarantee reasoning is a promising technique to address this challenge by reducing the verification problem to reasoning about each thread individually with the help of assertions about other threads. In this paper, we propose a proof rule that uses rely-guarantee reasoning for compositional verification of termination properties. The crux of our proof rule lies in its compositionality wrt. the thread structure of the program and wrt. the applied termination arguments – transition invariants. We present a method for automating the proof rule using an abstraction refinement procedure that is based on solving recursion-free Horn clauses. To deal with termination, we extend an existing Horn-clause solver with the capability to handle well-foundedness constraints. Finally, we present an experimental evaluation of our algorithm on a set of micro-benchmarks.

1 Introduction

Proving termination of various components of systems software is critical for ensuring the responsiveness of the entire system. Modern systems often contain multiple execution threads, yet most of the recent advances in automated termination proving for systems software focused on sequential programs, see e.g. [6,14,20]. Of course, in principle an existing termination prover for sequential programs can be applied to deal with non-cooperating threads by explicitly considering all possible thread interleavings, but such an approach is prohibitively expensive even for smallest programs.

Existing compositional methods for proving safety properties exploit thread structure to facilitate scalable reasoning and can deal with intricate thread interaction, see e.g. [2, 11, 12]. Unfortunately, these methods are not directly applicable for proving termination, since they rely on a finite-state abstraction for approximating the set of reachable program states [19].

Rely-guarantee reasoning [13] is a promising basis for the development of termination provers for multi-threaded programs. The method proposed in [7] relies on environment transitions that keep track of the interaction of a thread with its environment to prove termination properties of individual threads in a multi-threaded program. To ensure scalability, the underlying proof rule is

C. Flanagan and B. König (Eds.): TACAS 2012, LNCS 7214, pp. 237–251, 2012.

thread-modular and hence incomplete, i.e., it considers a restricted class of environment transitions that refer only to global variables. One practical consequence of such an incompleteness is that termination proofs of programs that use synchronization primitives like mutexes are out of scope. This limitation can be eliminated by gradually exposing additional local state when necessary [3]. Providing ranking functions for each thread that are preserved under environment transitions yields a complete method for proving termination that is compositional wrt. thread structure.

In this paper, we explore a combination of two dimensions of compositionality for proving termination of multi-threaded programs. We present a complete proof rule that is compositional wrt. the thread structure of the program and wrt. the termination argument that is used for each of the threads. Following the rely-guarantee reasoning approach, we use environment transitions to keep track of the effect of the threads from the environment of a given thread. As termination argument we rely on transition invariants and disjunctive well-foundedness [19], whose discovery can be efficiently automated in compositional fashion using abstract interpretation. The completeness of our proof rule is achieved by allowing the environment transitions to refer to both global and local variables of all threads. We also provide a specialized version of the proof rule for checking thread termination, i.e., termination of individual threads [7].

We demonstrate the potential for automation of our proof rule by transforming the recursive equations that represent the proof rule into a function whose least fixpoint characterizes the strongest proof of termination. We propose a transition predicate abstraction and refinement-based algorithm to obtain a sufficiently precise over-approximation of the least fixpoint and thus prove termination as follows. Since the fixpoint computation keeps track of both transition invariants and environment transitions, we obtain counterexamples that have a Horn-clause structure similar to a recent approach [11], yet generalizing from reachability to binary reachability. By analyzing the least solution to the counterexample we determine a well-founded over-approximation of discovered lasso-shaped thread interleavings. Given this over-approximation, we perform the actual transition predicate abstraction refinement by computing interpolating solutions to the Horn clauses. Technically, we extend the Horn clause solver presented in [11] with the treatment of well-foundedness constraints.

In summary, our paper makes the following contributions: i) a compositional proof rule that is complete for checking termination and thread termination of multi-threaded programs, ii) a method for automating the proof rule by using a corresponding predicate abstraction and refinement scheme, and iii) the implementation of our algorithm together with the evaluation of its feasibility on micro-benchmarks.

2 Preliminaries

Programs A *multi-threaded* program P consists of $N \geq 1$ threads. Let $1..N$ be the set $\{1, \ldots, N\}$. We assume that the *program* variables $V = (V_G, V_1, \ldots, V_N)$

are partitioned into *global* variables V_G, which are 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. Let $\rho_i^=$ stand for $V_i = V_i'$. Let $\models$ denote the satisfaction relation between (pairs) of states and assertions over program variables (and their primed versions). We use $\rightarrow$ as the logical implication operator as well as the logical consequence relation, and rely on the context for disambiguation.

The set of *initial* program states is denoted by φ_{init} . For each thread $i \in 1..N$ we have a finite set of *transitions* $\mathcal{R}_i$. Each transition is a binary relation between sets of program states. Furthermore, each $\rho \in \mathcal{R}_i$ 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), i.e., we have $\rho \rightarrow \rho_i^=$. We write ρ_i for the union of the transitions of the thread i , i.e., $\rho_i = \bigvee \mathcal{R}_i$. The transition relation of the program is $\rho_P = \rho_1 \vee \cdots \vee \rho_N$.

Computations. 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., $s_1 \models \varphi_{init}$, 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} denote the set of all reachable states. The program is *terminating* if it does not have any infinite computations. A *thread* i is *terminating*, as defined in [7], if there is no program computation that contains infinitely many transitions of i .

Auxiliary definitions. A binary relation φ is *well-founded* if it does not admit any infinite sequences. Let φ^+ denote the *transitive closure* of φ . A *transition invariant* T [19] is binary relation over program states that contains the transitive closure of the program transition relation restricted to reachable states, i.e., $\varphi_{reach} \wedge \rho_P^+ \rightarrow T$. Let $\varphi[z/w]$ denote a substitution that replaces w by z in φ . We assume that a sequence of substitutions is evaluated from left to right. Let $\circ$ be the *relational composition* function: $\varphi \circ \psi \;=\; \exists V'' : \varphi[V''/V'] \wedge \psi[V''/V]$. A *path relation* is a relational composition of transition relations along the path, i.e., for $\pi = \rho_1 \cdots \rho_n$ we have $\rho_\pi = \rho_1 \circ \ldots \circ \rho_n$. A path π is *feasible* if its path relation is not empty, i.e., $\exists V \exists V' : \rho_\pi$. Given a binary relation φ, we define an *image* function $Img(\varphi) = \exists V'' : \varphi[V''/V][V/V']$.

A *Horn clause* $b_1(w_1) \wedge \cdots \wedge b_n(w_n) \rightarrow b(w)$ consists of relation symbols $b_1, \ldots, b_n$, b, and vectors of variables $w_1, \ldots, w_n$, w. We say that a relation symbol b *depends* on the relation symbols $\{b_1, \ldots, b_n\}$. We distinguish interpreted theory symbols that we use to write assertions denoting sets (of pairs) of program states, e.g., the equality $=$ or the inequality $\leq$. A set of Horn clauses is *recursion-free* if it induces a well-founded dependency relation on non-interpreted relation symbols.

For assertions $T_1, \ldots, T_N$, $E_1, \ldots, E_N$ over V and V',
and well-founded relations $WF_1, \ldots, WF_m$

$$
\begin{array}{lll}
\text{C1:} & \varphi_{init} \wedge (\rho_i \vee E_i \wedge \rho_i^{=}) & \rightarrow T_i \quad \text{for } i \in 1..N \\
\text{C2:} & Img(T_i) \wedge (\rho_i \vee E_i \wedge \rho_i^{=}) & \rightarrow T_i \quad \text{for } i \in 1..N \\
\text{C3:} & T_i \circ (\rho_i \vee E_i \wedge \rho_i^{=}) & \rightarrow T_i \quad \text{for } i \in 1..N \\
\text{C4:} & (\bigvee_{i \in 1..N \setminus \{j\}} \varphi_{init} \wedge \rho_i) & \rightarrow E_j \quad \text{for } j \in 1..N \\
\text{C5:} & (\bigvee_{i \in 1..N \setminus \{j\}} Img(T_i) \wedge \rho_i) & \rightarrow E_j \quad \text{for } j \in 1..N \\
\text{C6:} & T_1 \wedge \cdots \wedge T_N & \rightarrow WF_1 \vee \cdots \vee WF_m
\end{array}
$$

multi-threaded program P terminates

Fig. 1. Proof rule ProgTerm for compositional proving of program termination

3 Proof Rules

In this section we present compositional rules for proving termination of multi-threaded programs and their threads.

3.1 Compositional Termination of Multi-threaded Programs

In order to reason about termination properties of multi-threaded programs, we propose a proof rule with two auxiliary assertions per thread denoted as T_i and E_i. $T_1, \ldots, T_N$ stand for transition invariants for respective threads. $E_1, \ldots, E_N$ represent environment transitions considered during the computation of transition invariants.

See Figure 1 for the proof rule ProgTerm. The first five premises ensure that T_i is a transition invariant of a program P as follows. The premises C1 and C2 require that T_i over-approximates the transition relation of the thread i restricted to initial states and to arbitrary reachable states. The same two premises require that T_i also over-approximates environment transitions $E_i \wedge \rho_i^{=}$. The conjunction with $\rho_i^{=}$ ensures that local variables of thread i do not change when an environment transition is applied. In the premise C3, extending a relation from T_i with either a local or an environment transition results in a relation that is also present in T_i. Two premises are used to record environment transitions for thread j that are induced by transitions executed by thread i either when starting from initial states (premise C4) or from arbitrary reachable states (premise C5). While the first five premises ensure the soundness of the transition invariants, the last premise C6 requires the existence of a disjunctive well-founded relation, e.g., $WF_1 \vee \cdots \vee WF_m$, as a witness for program termination.

Example 1. See Figure 2 for `Choice`, a multi-threaded version of the example with the same name from [19]. We use a parallel assignment instruction `(x,y)=...` to simultaneously update both variables `x` and `y`. Our verification method represents the program using assertions. The tuple $V = (\mathtt{x}, \mathtt{y}, \mathtt{pc}_1, \mathtt{pc}_2)$

```
int x,y;

// Thread T1                          // Thread T2
a0: while (x>0 && y>0) {              b0: while (x>0 && y>0) {
a1:   (x,y) = (x-1,x);                b1:   (x,y) = (y-2,x+1);
a2: }                                 b2: }
```

$$\varphi_{init} = (\mathtt{pc}_1 = \mathtt{a0} \wedge \mathtt{pc}_2 = \mathtt{b0})$$
$$\rho_1 = (\mathtt{x} > 0 \wedge \mathtt{y} > 0 \wedge \mathtt{x}' = \mathtt{x} - 1 \wedge \mathtt{y}' = \mathtt{x} \wedge \mathtt{pc}_1 = \mathtt{a0} \wedge \mathtt{pc}_1{}' = \mathtt{a0} \wedge \rho_2^{=})$$
$$\rho_2 = (\mathtt{x} > 0 \wedge \mathtt{y} > 0 \wedge \mathtt{x}' = \mathtt{y} - 2 \wedge \mathtt{y}' = \mathtt{x} + 1 \wedge \mathtt{pc}_2 = \mathtt{b0} \wedge \mathtt{pc}_2{}' = \mathtt{b0} \wedge \rho_1^{=})$$

Fig. 2. Example program `T1 || T2` for which the value of either of the following expressions decreases: x, y or $x + y$. We use $\rho_1^{=} = (\mathtt{pc}_1 = \mathtt{pc}_1{}')$ and $\rho_2^{=} = (\mathtt{pc}_2 = \mathtt{pc}_2{}')$

includes $\mathtt{pc}_1$ and $\mathtt{pc}_2$, the program counter variables of the two threads. φ_{init} describes the initial states of the program, while ρ_1 and ρ_2 represent the transition relations of the two threads (simplified for clarity of illustration).

We show that `Choice` terminates by applying ProgTerm and considering the following auxiliary assertions.

$$\begin{aligned} T_1 = T_2 = (&\mathtt{x} > 0 \wedge \mathtt{y} > 0 \wedge \mathtt{x}' \leq \mathtt{x} - 1 \wedge \mathtt{y}' \leq \mathtt{x} + 1) \vee \\ &(\mathtt{x} > 0 \wedge \mathtt{y} > 0 \wedge \mathtt{x}' \leq \mathtt{y} - 2 \wedge \mathtt{y}' \leq \mathtt{y} - 1) \\ E_1 = (&\mathtt{x} > 0 \wedge \mathtt{y} > 0 \wedge \mathtt{x}' \leq \mathtt{y} - 2 \wedge \mathtt{y}' \leq \mathtt{x} + 1) \\ E_2 = (&\mathtt{x} > 0 \wedge \mathtt{y} > 0 \wedge \mathtt{x}' \leq \mathtt{x} - 1 \wedge \mathtt{y}' \leq \mathtt{x}) \end{aligned}$$

The assertion E_1 approximates the effect of applying the transition from the second thread, while E_2 approximates the effect of applying the transition from the first thread. There exists a disjunctively well-founded relation, $(\mathtt{x} > \mathtt{x}' \wedge \mathtt{x} \geq 0 \vee \mathtt{y} > \mathtt{y}' \wedge \mathtt{y} \geq 0)$, that approximates the transition invariants T_1 and T_2 .

This example also illustrates a limitation of the proof rule from [7], which requires that the environment transitions must be transitive. Our non-transitive environment transitions can be weakened to their transitive closure, but then the transitive closure E_1^+ is too weak to prove termination since the constraints $\mathtt{x}' \leq \mathtt{y} - 2$ and $\mathtt{y}' \leq \mathtt{x} + 1$ will be missing.

3.2 Compositional Thread Termination

For the cases where the program termination is a property too strong to hold (e.g., dispatch routines of event-based systems continuously accept incoming events), it is still the case that termination of critical threads is important [7]. See Figure 3 for a proof rule that relaxes the conditions from ProgTerm but still ensures the thread-termination property for some thread from a given program.

The rule ThreadTerm relies on assertions T_k and E_k that satisfy the first five premises from the rule ProgTerm. For a thread k , an additional assertion $\hat{t}_k$ is used to keep track of a subset of the transition invariant relation T_k . Unlike

For assertions $T_1, \ldots, T_N$, $E_1, \ldots, E_N$, $\hat{t}_k$ over V and V'
that satisfy C1, C2, C3, C4, C5 and well-founded relations $WF_1, \ldots, WF_m$

$$\begin{array}{lll} \text{C1':} & \varphi_{init} \wedge \rho_k & \rightarrow \hat{t}_k \\ \text{C2':} & Img(T_k) \wedge \rho_k & \rightarrow \hat{t}_k \\ \text{C3':} & \hat{t}_k \circ (\rho_k \vee E_k \wedge \rho_k^{=}) & \rightarrow \hat{t}_k \\ \text{C6':} & \hat{t}_k & \rightarrow WF_1 \vee \cdots \vee WF_m \end{array}$$

thread k terminates

Fig. 3. Proof rule ThreadTerm for compositional proving of thread termination

the program-termination premises C1 and C2, the thread-termination premises C1' and C2' require that transitions captured by $\hat{t}_k$ always start with a local transition from thread k . With this restriction in place, the premise C3' is similar to C3: it extends $\hat{t}_k$ with either a local transition ρ_k or an environment transition. The assertion $\hat{t}_k$ keeps track of thread interleavings that may lead to computations of arbitrary length, and its disjunctive well-foundedness is required for thread-termination of thread k .

4 Proof Rule Automation

Next, we demonstrate the potential for automation of the proof rule ProgTerm. (Automation of the proof rule ThreadTerm is similar.) We formulate an algorithm for proving program termination that consists of three steps. The first step uses abstraction functions to compute transition invariants for each thread. If this process discovers abstract transitions – components of transition invariants – whose intersection is not disjunctively well-founded, then the second step generates Horn clauses such that their satisfiability implies a disjunctively well-founded refined counterpart exists. The last step of the algorithm invokes a solving procedure for the Horn clauses and uses the obtained solution to refine the abstraction functions.

The entry point of our algorithm initializes the transition abstraction functions using empty sets of predicates. Then, it repeats a loop that invokes transition invariant computation that is followed by an abstraction refinement step.

Procedure AbstTransEnv. Figure 4 shows the reachability procedure that computes abstract transitions $\mathcal{T}_i$ and abstract environment transitions $\mathcal{E}_i$ following the conditions from the ProgTerm proof rule, where $\mathcal{T}_i$ and $\mathcal{E}_i$ correspond to auxiliary assertions T_i and E_i, respectively. These sets are initialized corresponding with the rules C1 and C4 in lines 1–4. The auxiliary procedure AddIfNew implements a fixpoint check. It takes as input a newly computed binary relation τ and a container set $\mathcal{C}$. It checks if the binary relation contains pairs of states that have not been reached and recorded in the container. If this check succeeds, the new binary relation is added to the container and the

```
   global variables
     P - program with N threads
     P̈_i, ä_i, P̈_{i▷j}, ä_{i▷j} - transition predicates and transition abstraction functions
     T_i, E_i - abstract (environment) transitions of thread i
     Parent, ParentTId - parent relations
   procedure AbstTransEnv
   begin
1    for each i ∈ 1..N and ρ ∈ R_i do
2      AddIfNew(ä_i(φ_init ∧ ρ), T_i, (φ_init, ρ), i)                    (* C1 *)
3      for each j ∈ 1..N \ {i} do
4        AddIfNew(ä_{i▷j}(φ_init ∧ ρ), E_j, (φ_init, ρ), i)              (* C4 *)
5    repeat
6      finished := true
7      for each i ∈ 1..N and τ ∈ T_i do
8        for each ρ ∈ R_i ∪ E_i do
9          φ := if ρ ∈ R_i then true else ρ_i^=
10         AddIfNew(ä_i(Img(τ) ∧ ρ ∧ φ), T_i, (Img(τ), ρ), i)            (* C2 *)
11         AddIfNew(ä_i(τ ∘ (ρ ∧ φ)), T_i, (τ, ρ), i)                   (* C3 *)
12       for each ρ ∈ R_i and j ∈ 1..N \ {i} do
13         τ' := ä_{i▷j}(Img(τ) ∧ ρ)
14         AddIfNew(τ', E_j, (Img(τ), ρ), i)                           (* C5 *)
15         AddIfNew(ä_j(φ_init ∧ τ' ∧ ρ_j^=), T_j, (φ_init, τ'), j)    (* C1 *)
16   until finished
   end
   procedure AddIfNew
   input
     τ - binary relation
     C - container set, either T_i or E_i for some i ∈ 1..N
     P - parent pair
     j - thread identifier from 1..N
   begin
     if ¬(∃ρ ∈ C : τ → ρ) then
17     C := {τ} ∪ C
18     Parent(τ) := P
19     ParentTId(τ) := j
20     finished := false
21 end
```

Fig. 4. Procedure AbstTransEnv computes abstract transitions $\mathcal{T}_i$ and $\mathcal{E}_i$. It uses a local auxiliary procedure AddIfNew

Parent function is updated to keep track of the child-parent relation between transitions.

The second part of the procedure (see lines 5–16) ensures that the other conditions from the proof rule are satisfied. It computes an abstract reachable state using the $Img(\tau)$ operator and then applies the abstract transition ρ from this abstract state. The result is extended with a transition local to thread i or from its environment in line 10. Next, one-step environment transitions are generated and added both to $\mathcal{E}_j$ in line 14 and $\mathcal{T}_j$ in line 15. Whenever an

procedure TermRefine
input
$\tau_1, \ldots, \tau_N$ - abstract error tuple
begin
1 $HC := \text{MkHC}(\tau_1) \cup \cdots \cup \text{MkHC}(\tau_N)$
2 $UnkRels := \{\text{“}\rho\text{”}(V, V') \mid i \in 1..N \wedge \rho \in \mathcal{T}_i \cup \mathcal{E}_i\}$
3 $UnkRelsWF := \{\text{“}\tau_1\text{”}(V, V'), \ldots, \text{“}\tau_N\text{”}(V, V')\}$
4 $\text{Sol} := \text{SolveHC}^{WF}(HC, UnkRelsWF, UnkRels))$
5 **for** each $i \in 1..N$ and $\rho \in \mathcal{T}_i$ **do**
6 $\ddot{\mathcal{P}}_i := PredsOf(\text{Sol}(\text{“}\rho\text{”}(V, V'))) \cup \ddot{\mathcal{P}}_i$
7 **for** each $j \in 1..N$ and $\rho \in \mathcal{E}_j$ **do**
8 $i := ParentTId(\rho)$
9 $\ddot{\mathcal{P}}_{i \triangleright j} := PredsOf(\text{Sol}(\text{“}\rho\text{”}(V, V'))) \cup \ddot{\mathcal{P}}_{i \triangleright j}$
end

procedure SolveHC^{WF}
input
HC - recursion-free Horn clauses
$UnkRelsWF = \{\text{“}\tau_1\text{”}(V, V'), \ldots, \text{“}\tau_N\text{”}(V, V')\}$,
$UnkRels$ - unknown relations
output
Sol - solution for HC such that $UnkRelsWF$ are well-founded relations
begin
10 $\text{Sol}^{\mu} := \text{SolveHC}^{\mu}(HC, UnkRelsWF)$
11 $\rho := \text{Sol}^{\mu}(\text{“}\tau_1\text{”}(V, V')) \wedge \cdots \wedge \text{Sol}^{\mu}(\text{“}\tau_N\text{”}(V, V'))$
12 **if** $\exists WF : well\text{-}founded(WF) \wedge (\rho \rightarrow WF)$ **then**
13 $HC^{WF} := \{\text{“}\tau_1\text{”}(V, V') \wedge \cdots \wedge \text{“}\tau_N\text{”}(V, V') \rightarrow WF\} \cup HC$
14 $\text{Sol} := \text{SolveHC}(HC^{WF}, UnkRels)$
15 **else**
16 **throw** Unsatisfiable
end

Fig. 5. Procedure TermRefine takes as argument an abstract error tuple. The quotation function “ · ” creates a relation symbol from a given abstract (environment) transition. The function *PredsOf* extracts atomic predicates from the solutions to the set of Horn clauses HC , while MkHC generates Horn clauses

iteration of the **repeat** loop finishes with the *finished* flag set to *true*, the proof rules are saturated and the procedure returns a fixpoint for $\mathcal{T}_i$ and $\mathcal{E}_i$.

Procedure TermRefine. See Figure 5 for the procedure that takes as input a tuple of abstract transitions and computes, if possible, predicates that witness the well-foundedness of these transitions. The procedure generates in line 1 a set of Horn clauses corresponding to the abstract transitions. It collects in the set *UnkRels* unknown relations corresponding to each abstract (environment) transition. The set *UnkRelsWF* is a subset of *UnkRels* that contains unknown relations with an additional requirement: the conjunction of their solutions must be a well-founded relation.

After invoking the solving procedure at line 4, the solution SOL is used to update the transition abstraction functions as follows. Atomic predicates from SOL("ρ"(V, V')) are added to the set of predicates $\ddot{\mathcal{P}}_i$ corresponding to the parent thread of ρ . Similarly, SOL("ρ"(V, V')) is used to update $\ddot{\mathcal{P}}_{i \triangleright j}$ if ρ is an environment transition that was constructed in thread i and applied in thread j . These new predicates guarantee that the same counterexample will not be encountered during the next iterations of reachability analysis.

Procedure SOLVEHC^{WF}. Figure 5 shows a procedure for solving recursion-free Horn clauses such that the solution for some unknown relations is well-founded.

Due to the recursion-free nature of the Horn clauses, computing the least (most precise) solution for the clauses is trivial. Note that the set of Horn clauses HC is guaranteed to have at least the solution mapping every unknown to a *true* predicate. At line 10, the least solution SOL$^{\mu}$ that is returned is defined for the unknown relations from $UnkRels_{WF}$. If the solution SOL$^{\mu}$ does not have an upper-bound that is well-founded (see line 12), then the abstraction refinement fails with an UNSATISFIABLE exception. The test at line 12 can be implemented using a rank synthesis tool, e.g. [18].

If a well-founded relation WF is detected, then an additional Horn clause ensures that the solution for the abstract transitions over-approximates it: $\{$"τ_1"$(V, V') \wedge \cdots \wedge$ "τ_N"$(V, V') \rightarrow WF\}$. Given the set of Horn clauses HC^{WF} , our algorithm invokes a solving procedure in line 9, e.g., SOLVEHC from [11].

To ensure the completeness of the refinement procedure (relative to the domain of linear inequalities), solutions to unknown relations "ρ"(V, V') are expressed in terms of all program variables V and their primed version V'. However, to facilitate thread-modular reasoning, it is desirable to return solutions for transitions in $\mathcal{T}_i$ that are expressed over V_G, V_i and solutions for environment transitions in $\mathcal{E}_i$ over global program variables V_G . The Horn clause solving procedure SOLVEHC ensures that, whenever it exists, a modular solution is returned. Both reasoning about safety [11] and reasoning about termination properties benefit from such modular solutions.

Example 2. The `LockDecrement` example was used to illustrate the thread-modular verification method from [7]. Our algorithm is able to verify a termination property for this example even if we drop two of the assumptions required by the verification method from [7].

The code for this example is shown in Figure 6 and consists of two threads `T1` and `T2`. The thread `T1` acquires a lock on line `a0` and decrements the value of the variable `x` if it is positive. The thread `T2` assigns a non-deterministically chosen value to `x` after it acquires the lock `lck` on line `b1` . The program assumes that initially the lock is not taken, i.e., `lck` $= 0$. The locking statement `lock(lck)` waits until the lock is released and then assigns the value 1 to `lck` , thus taking the lock. We are interested in proving thread-termination of the first thread `T1`, i.e., no computation of the program contains infinitely many steps from `T1`. Note that the non-termination of the thread `T2` (and of the entire program `T1 || T2`) does not affect the thread-termination property of `T1`.

```
// Thread T1
a0: lock(lck);
a1: while (x>0) {
a2:   t = x-1;
a3:   x = t;
a4: }
a5: unlock(lck);
a6:
```

$$\varphi_{init} = (\mathtt{lck} = 0 \wedge \mathtt{pc}_1 = \mathtt{a0} \wedge \mathtt{pc}_2 = \mathtt{b0})$$
$$\rho_{11} = (\mathtt{lck} = 0 \wedge \mathtt{lck}' = 1 \wedge \mathit{skip}(\mathtt{x},\mathtt{t}) \wedge \mathit{move}_1(\mathtt{a0},\mathtt{a1}) \wedge \rho_2^{=})$$
$$\rho_{12} = (\mathtt{x} > 0 \wedge \mathit{skip}(\mathtt{x},\mathtt{lck},\mathtt{t}) \wedge \mathit{move}_1(\mathtt{a1},\mathtt{a2}) \wedge \rho_2^{=})$$
$$\rho_{13} = (\mathtt{t}' = \mathtt{x} - 1 \wedge \mathit{skip}(\mathtt{x},\mathtt{lck}) \wedge \mathit{move}_1(\mathtt{a2},\mathtt{a3}) \wedge \rho_2^{=})$$
$$\rho_{14} = (\mathtt{x}' = \mathtt{t} \wedge \mathit{skip}(\mathtt{lck},\mathtt{t}) \wedge \mathit{move}_1(\mathtt{a3},\mathtt{a1}) \wedge \rho_2^{=})$$
$$\rho_{15} = (\mathtt{x} \leq 0 \wedge \mathit{skip}(\mathtt{x},\mathtt{lck},\mathtt{t}) \wedge \mathit{move}_1(\mathtt{a1},\mathtt{a5}) \wedge \rho_2^{=})$$
$$\rho_{16} = (\mathtt{lck}' = 0 \wedge \mathit{skip}(\mathtt{x},\mathtt{t}) \wedge \mathit{move}_1(\mathtt{a5},\mathtt{a6}) \wedge \rho_2^{=})$$
$$\rho_2^{=} = (\mathtt{pc}_2{}' = \mathtt{pc}_2)$$

```
// Thread T2
b0: while (nondet()) {
b1:   lock(lck);
b2:   x = nondet();
b3:   unlock(lck);
b4: }
```

$$\rho_{21} = (\mathtt{lck} = 0 \wedge \mathtt{lck}' = 1 \wedge \mathit{skip}(\mathtt{x}) \wedge \mathit{move}_2(\mathtt{b0},\mathtt{b2}) \wedge \rho_1^{=})$$
$$\rho_{22} = (\mathit{skip}(\mathtt{lck}) \wedge \mathit{move}_2(\mathtt{b2},\mathtt{b3}) \wedge \rho_1^{=})$$
$$\rho_{23} = (\mathtt{lck}' = 0 \wedge \mathit{skip}(\mathtt{x}) \wedge \mathit{move}_2(\mathtt{b3},\mathtt{b0}) \wedge \rho_1^{=})$$
$$\rho_{24} = (\mathit{skip}(\mathtt{x},\mathtt{lck}) \wedge \mathit{move}_2(\mathtt{b0},\mathtt{b5}) \wedge \rho_1^{=})$$
$$\rho_1^{=} = (\mathtt{t}' = \mathtt{t} \wedge \mathtt{pc}_1{}' = \mathtt{pc}_1)$$

Fig. 6. Example with global variables $\mathtt{x}, \mathtt{t}, \mathtt{lck}$, with `lck` initially set to 0 (unlocked)

See Figure 6 for the program representation, where $\rho_{11}, \ldots, \rho_{16}$ give the transition relation of `T1`, while $\rho_{21}, \ldots, \rho_{24}$ represent the transition relation of `T2`. The notation $\mathit{skip}(\mathtt{x},\mathtt{t})$ is a shorthand for a constraint indicating that the values of `x` and `t` are not changed, i.e., $(\mathtt{x} = \mathtt{x}' \wedge \mathtt{t} = \mathtt{t}')$, while $\mathit{move}_1(\mathtt{a0},\mathtt{a1})$ denotes that the location of thread `T1` is changed from `a0` to `a1`, i.e., $(\mathtt{pc}_1 = \mathtt{a0} \wedge \mathtt{pc}_1{}' = \mathtt{a1})$.

Next, we illustrate how a termination proof for the `LockDecrement` program can be constructed automatically using a sequence of abstract reachability and abstraction refinement steps.

First abstract computation. The abstract computation approximates the transition relation of the program using two abstraction functions $\ddot{\alpha}_1$ and $\ddot{\alpha}_2$ that correspond to threads `T1` and respectively `T2`. For this example, we assume that the abstraction functions initially track the value of the program counters of the two threads using the following sets of predicates: $\ddot{\mathcal{P}}_1 = \{\mathtt{pc}_1 = \mathtt{a0}, \ldots, \mathtt{pc}_1{}' = \mathtt{a0}, \ldots\}$ and $\ddot{\mathcal{P}}_2 = \{\mathtt{pc}_2 = \mathtt{b0}, \ldots, \mathtt{pc}_2{}' = \mathtt{b0}, \ldots\}$. The abstraction of a transition relation is computed as $\ddot{\alpha}_i(T) = \bigwedge\{\ddot{p} \in \ddot{\mathcal{P}}_i \mid T \rightarrow \ddot{p}\}$. Our algorithm starts by computing one-step relations from initial states (see premise C1). We obtain two abstract transitions: $m_1 = \ddot{\alpha}_1(\varphi_{init} \wedge \rho_{11}) = (\mathtt{pc}_1 = \mathtt{a0} \wedge \mathtt{pc}_1{}' = \mathtt{a1})$ and $n_1 = \ddot{\alpha}_2(\varphi_{init} \wedge \rho_{21}) = (\mathtt{pc}_2 = \mathtt{b0} \wedge \mathtt{pc}_2{}' = \mathtt{b2})$. The abstract transition m_1 groups sequences of concrete transitions that start with $\mathtt{pc}_1 = \mathtt{a0}$ and finish with $\mathtt{pc}_1 = \mathtt{a1}$, e.g., the sequence of four transitions $\rho_{11}\rho_{12}\rho_{13}\rho_{14}$.

Next, our algorithm computes successors for the abstract transitions that were already discovered (see premise C2). For m_1 , we compute the image of the relation using the operator $\mathit{Img}(m_1) = (\mathtt{pc}_1 = \mathtt{a1})$. Then, we compute a successor with regards to a transition ρ_{12} as follows: $m_2 = \ddot{\alpha}_1(\mathit{Img}(m_1) \wedge \rho_{12}) = (\mathtt{pc}_1 = \mathtt{a1} \wedge \mathtt{pc}_1{}' = \mathtt{a2})$. A second method to derive successors for an abstract transition is to extend it with another transition from the thread transition

relation (see premise C3). For m_2 , we compute its extension using the relational composition operator, $m_2 \circ \rho_{13} = (\mathtt{pc}_1 = \mathtt{a1} \wedge \mathtt{pc}_1{}' = \mathtt{a2}) \circ (\mathtt{x}' = \mathtt{x} \wedge \mathtt{lck}' = \mathtt{lck} \wedge \mathtt{t}' = \mathtt{x} - 1 \wedge \mathtt{pc}_1 = \mathtt{a2} \wedge \mathtt{pc}_1{}' = \mathtt{a3} \wedge \rho_2^{=}) = (\mathtt{pc}_1 = \mathtt{a1} \wedge \mathtt{pc}_1{}' = \mathtt{a3})$. From m_2, two new abstract transitions are generated: $m_3 = \ddot{\alpha}_1(m_2 \circ \rho_{13}) = (\mathtt{pc}_1 = \mathtt{a1} \wedge \mathtt{pc}_1{}' = \mathtt{a3})$ and $m_4 = \ddot{\alpha}_1(m_3 \circ \rho_{14}) = (\mathtt{pc}_1 = \mathtt{a1} \wedge \mathtt{pc}_1{}' = \mathtt{a1})$.

The abstract computation continues until no new abstract transitions are found or there is an indication of non terminating executions. For our example, the abstract transition m_4 is not well-founded, i.e., an infinite sequence of such transitions appears to be feasible.

First abstraction refinement. Since m_4 was obtained using abstraction, we need to check whether the evidence for non-termination is spurious. This is realized by computing a constraint that is satisfiable if and only if the abstraction can be refined to remove the spurious transition. This constraint portrays the way the abstract transition m_4 was computed using unknown predicates to denote abstract transitions. Since m_1 was computed as an abstraction from the initial state, $\varphi_{init} \wedge \rho_{11} \rightarrow m_1(V, V')$, the constraint maintains this requirement by replacing the abstract transition with an unknown predicate that denotes the abstract transition to be refined: $\varphi_{init} \wedge \rho_{11} \rightarrow \text{“}m_1\text{”}(V, V')$. The constraint is represented by a set of such implications, each one in the form of a Horn clause (with implicit universal quantifiers). We obtain HC_1 as follows.

$$\begin{aligned} HC_1 = \{ & \varphi_{init} \wedge \rho_{11} \rightarrow \text{“}m_1\text{”}(V, V'), \\ & \text{“}m_1\text{”}(V'', V) \wedge \rho_{12} \rightarrow \text{“}m_2\text{”}(V, V'), \\ & \text{“}m_2\text{”}(V'', V) \wedge \rho_{13} \rightarrow \text{“}m_3\text{”}(V'', V'), \\ & \text{“}m_3\text{”}(V'', V) \wedge \rho_{14} \rightarrow \text{“}m_4\text{”}(V'', V') \ \} \end{aligned}$$

The constraint HC_1 may have multiple solutions for the unknown predicates. Our algorithm uses two types of solutions for HC_1 . First, we need to ensure that there exists a well-founded transition that over-approximates $\text{“}m_4\text{”}(V, V')$. This is done by computing the least (most precise) solution for $\text{“}m_4\text{”}(V, V')$ over some domain of solutions. For our example, we obtain Sol^{μ} .

$$\begin{aligned} \text{Sol}^{\mu}(\text{“}m_1\text{”}(V, V')) &= (\ \mathtt{x}' = \mathtt{x} \wedge \mathtt{lck} = 0 \wedge \mathtt{lck}' = 1 \wedge \mathtt{t}' = \mathtt{t}) \\ \text{Sol}^{\mu}(\text{“}m_2\text{”}(V, V')) &= (\ \mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{lck} = \mathtt{lck}' = 1 \wedge \mathtt{t}' = \mathtt{t}) \\ \text{Sol}^{\mu}(\text{“}m_3\text{”}(V, V')) &= (\ \mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{lck} = \mathtt{lck}' = 1 \wedge \mathtt{t}' = \mathtt{x} - 1) \\ \text{Sol}^{\mu}(\text{“}m_4\text{”}(V, V')) &= (\ \mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} - 1 \wedge \mathtt{lck} = \mathtt{lck}' = 1 \wedge \mathtt{t}' = \mathtt{x} - 1) \end{aligned}$$

The solution $\text{Sol}^{\mu}(\text{“}m_4\text{”}(V, V'))$ is well-founded. While refining the abstraction functions using the solution Sol^{μ} would remove the spurious transition that is not well-founded, such a solution hinders the convergence of the abstraction refinement procedure. For the current example, collecting all predicates that appear in the least solutions of $\text{“}m_1\text{”}(V, V')$, $\text{“}m_2\text{”}(V, V')$, $\text{“}m_3\text{”}(V, V')$ and $\text{“}m_4\text{”}(V, V')$ would compromise the benefit of abstraction. Instead, our algorithm uses the method from [18] to generate a well-founded relation that approximates

the transitive closure of $\text{SOL}^{\mu}(\text{“}m_4\text{”}(V,V')) : WF_1(V,V') = (\mathtt{x}' \geq 0 \wedge \mathtt{x}' < \mathtt{x})$. The next step is to add a well-foundedness condition to the set of Horn clauses $HC_1^{WF} = HC_1 \cup \{\text{“}m_4\text{”}(V,V') \rightarrow WF_1\}$, and look for a solution of the extended set of clauses. We obtain the solution SOL by invoking the solving algorithm from [11] with input HC_1^{WF} .

$$\text{SOL}(\text{“}m_1\text{”}(V,V')) = (true) \qquad \text{SOL}(\text{“}m_2\text{”}(V,V')) = (\mathtt{x}' \geq 1)$$
$$\text{SOL}(\text{“}m_3\text{”}(V,V')) = (\mathtt{t}' \geq 0 \wedge \mathtt{t}' < \mathtt{x}) \quad \text{SOL}(\text{“}m_4\text{”}(V,V')) = (\mathtt{x}' \geq 0 \wedge \mathtt{x}' < \mathtt{x})$$

We collect the predicates that appear in the solutions of the abstract transitions from the first thread and add them to the set of predicates $\ddot{\mathcal{P}}_1$ as follows: $\ddot{\mathcal{P}}_1 = \ddot{\mathcal{P}}_1 \cup \{\mathtt{x}' \geq 1, \mathtt{t}' \geq 0, \mathtt{t}' < \mathtt{x}, \mathtt{x}' \geq 0, \mathtt{x}' < \mathtt{x}\}$.

Termination property of thread `T2`. We now illustrate the capability of our algorithm to identify termination bugs. The abstract computation of the second thread proceeds as follows: $n_1 = \ddot{\alpha}_2(\varphi_{init} \wedge \rho_{21}) = (\mathtt{pc_2} = \mathtt{b0} \wedge \mathtt{pc_2}' = \mathtt{b2})$, $n_2 = \ddot{\alpha}_2(n_1 \circ \rho_{22}) = (\mathtt{pc_2} = \mathtt{b0} \wedge \mathtt{pc_2}' = \mathtt{b3})$ and $n_3 = \ddot{\alpha}_2(n_2 \wedge \rho_{23}) = (\mathtt{pc_2} = \mathtt{b0} \wedge \mathtt{pc_2}' = \mathtt{b0})$. The abstract transition n_3 is not well-founded. Our algorithm generates the following set of three Horn clauses:

$$HC_4 = \{\varphi_{init} \wedge \rho_{21} \rightarrow \text{“}n_1\text{”}(V,V'), \text{“}n_1\text{”}(V,V') \wedge \rho_{22}(V',V'') \rightarrow \text{“}n_2\text{”}(V,V''),$$
$$\text{“}n_2\text{”}(V,V') \wedge \rho_{23}(V',V'') \rightarrow \text{“}n_3\text{”}(V,V'')\ \} .$$

The least solution $\text{SOL}^{\mu}(\text{“}n_3\text{”}(V,V')) = (\mathtt{lck} = 0 \wedge \mathtt{lck}' = 0 \wedge \mathtt{pc_1} = \mathtt{a0} \wedge \mathtt{pc_1}' = \mathtt{a0} \wedge \mathtt{pc_2} = \mathtt{b0} \wedge \mathtt{pc_2}' = \mathtt{b0})$ is not well-founded. The algorithm returns a counterexample to termination of thread T2, the sequence of transitions $\rho_{21}\rho_{22}\rho_{23}$.

Termination property of thread `T1`. After a few more iterations, our algorithm proves the termination of the first thread. It constructs all abstract transitions applying exhaustively the steps above and it finds that all of them represent well-founded abstract transitions. The counterexample π is not encountered during the proof of thread-termination of `T1`, since it contains only transitions from the second thread. We conclude the presentation of this example by showing the final environment transitions that are part of the proof for termination of the first thread. They ensure two conditions: 1) environment transitions constructed from ρ_{22} are guarded by the relation $\mathtt{pc_2} = \mathtt{b2}$, 2) environment transitions generated from other transitions of the second thread satisfy the constraint $\mathtt{x}' \leq \mathtt{x}$.

$$\ddot{\mathcal{P}}_{2\triangleright 1} = \{\mathtt{x}' \leq \mathtt{x}, \mathtt{lck} = 0, \mathtt{lck}' = 1, \mathtt{pc_2} = \mathtt{b2}, \mathtt{pc_2}' \neq \mathtt{b2}\}$$
$$E_1 = (\mathtt{pc_2} = \mathtt{b2} \wedge \mathtt{pc_2}' \neq \mathtt{b2}) \vee (\mathtt{pc_2}' \neq \mathtt{b2} \wedge \mathtt{x}' \leq \mathtt{x}) \vee$$
$$(\mathtt{lck} = 0 \wedge \mathtt{lck}' = 1 \wedge \mathtt{x}' \leq \mathtt{x})$$

The termination property of T1 cannot be established using the method proposed in [7], which is restricted to thread-modular proofs. The program `LockDecrement` lifts two restrictions from the original example [7]: the decrement of the value of `x` (not an atomic operation) is split in two statements at labels `a2` and `a3` ; secondly, instead of an integer, the value of the lock variable is represented using a single bit, as our method supports mutexes.

Table 1. Experimental evaluation

Benchmark programs		Proving Termination			
Name	LOC	Thread 1		Thread 1+2	Comments
Figure 2 [7]	19	✓-Modular	0.1s	×	T2 is non-term.
Figure 6	21	✓-NonModular	0.4s	×	T2 is non-term.
Figure 2	9	✓-Modular	5.9s	✓-Modular 9.4s	T1+T2 proven term.
Figure 2-fixed [15]	38	✓-Modular	0.3s	×	T2 is non-term.
Figure 4-fixed [15]	168	✓-Modular	13.2s	×	T2 is non-term.

5 Experiments

We implemented the algorithms for proving termination properties in a tool for verification of multi-threaded C programs. Our tool uses a frontend based on CIL [16] to translate C programs to multi-threaded transition systems as formalized in Section 2. As explained before, our implementation makes use of a procedure for finding ranking functions [18] and a Horn clause solving algorithm over the linear inequality domain [11]. As an optimization, we use the frontend capability of generating a cutpoint for every iterating construct in the input program. The implementation only checks for well-foundedness of transitions with start and end locations compatible with a cutpoint.

We evaluated our implementation using a set of benchmark programs listed in Table 1. The first example was presented as Figure 2 in [7]. Relying on an atomicity assumption encoded in the program, our tool is able to find a modular proof for the thread-termination of Thread 1. The second thread contains a non-terminating loop and our tool returns a counterexample to its termination.

The second program (see Figure 6) removes the atomicity assumption from the previous program and represents the lock variable using a single bit (i.e., a mutex). Under these assumptions, the termination of thread 1 has a non-modular proof that our tool is able to discover automatically.

For the example shown in Figure 2, our tool is able to derive a modular proof for the termination of both threads. This is facilitated by the abstraction refinement queries formulated in terms of Horn clauses with preference for modular solutions. More elaborate non-modular proofs of termination also exists for this program and our tool is able to avoid them.

We include two more examples that illustrate fixes from the Mozilla CVS repository for two vulnerabilities initially presented in a study of concurrency bugs [15]. The termination of the second thread from Figure 2-fixed depends on a fairness assumption and our tool reports a counterexample that represents an unfair path. In practice, fair termination is a desirable property, since it allows one to exclude from consideration pathological, unfair schedulers. Our approach can be extended to deal with fair termination via program transformation [17]. Fairness assumptions can be explicated in the program using additional counters and hence termination proving methods apply out of the box.

6 Related Work

Our paper builds on two lines of research. The first one established abstraction refinement [1] as a successful method for proving termination of sequential programs [5, 6]. Our second inspiration is the rely-guarantee method that was proposed in [13] for reasoning about multi-threaded programs. This reasoning method was automated and implemented for the first time in the Calvin model checker [8] for Java programs. Calvin's implementation makes use of environment assumptions specified by the programmer for each thread. More recently, discovery of environment assumptions for proving safety properties was automated using abstraction refinement [11]. Their abstraction refinement is based on a Horn-clause solving procedure, which we use in this paper for a different class of refinement queries (related to termination properties).

For proving termination of multi-threaded programs, Cook et al. proposed an automated method based on rely-guarantee reasoning [7]. This verification method was the first to compute environment assumptions for termination, but is restricted to properties with modular proofs and does not support synchronization primitives like mutexes. Cohen and Namjoshi present an algorithm that gradually exposes additional local state that is needed in non thread-modular safety proofs [2] and termination proofs [3]. Although their proof rules and general algorithms apply to non-finite analysis domains, they have so far been implemented and tested only for finite-state systems. Unlike our proposal that does not directly support fairness assumptions, a recent extension of the SPLIT model checker [4] incorporates fairness in a compositional algorithm without relying on the trivial solution that turns all local variables from the specification in global variables.

Recent work on compositional transition invariants deals with verification of sequential programs [14]. In this work, compositionality refers to the transitivity property of transition invariants and allows one to check the validity of a transition invariant candidate without going through the transitive closure computation. This notion of compositionality was successfully used to speed up the verification of sequential programs but is not applicable for structuring the verification of multi-threaded programs. In contrast, in our work and in the literature on the verification of multi-threaded programs it is common to use compositionality to describe verification methods that consider only one thread at a time (instead of taking the entire program, which is prohibitively expensive even for very small programs).

Thread-modular shape analysis [9] has been proposed for verification of heap-manipulating multi-threaded programs. The original formulation handled safety properties, while a follow-up work [10] proposed an automatic method for proving liveness properties of concurrent data-structure implementations. While this line of work handles only properties that have thread-modular proofs, we believe that the proof rule that we propose in this paper is a step towards automatic proving of properties that require non-modular reasoning about heap.

Acknowledgement. We thank Ruslán Ledesma-Garza and Kedar Namjoshi for comments and suggestions.

References

1. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
2. Cohen, A., Namjoshi, K.S.: Local Proofs for Global Safety Properties. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 55–67. Springer, Heidelberg (2007)
3. Cohen, A., Namjoshi, K.S.: Local Proofs for Linear-Time Properties of Concurrent Programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 149–161. Springer, Heidelberg (2008)
4. Cohen, A., Namjoshi, K.S., Sa'ar, Y.: A Dash of Fairness for Compositional Reasoning. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 543–557. Springer, Heidelberg (2010)
5. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction Refinement for Termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
6. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
7. Cook, B., Podelski, A., Rybalchenko, A.: Proving thread termination. In: PLDI (2007)
8. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-Modular Verification for Shared-Memory Programs. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 262–277. Springer, Heidelberg (2002)
9. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI (2007)
10. Gotsman, A., Cook, B., Parkinson, M.J., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: POPL (2009)
11. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL (2011)
12. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: PLDI (2004)
13. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)
14. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination Analysis with Compositional Transition Invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
15. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS (2008)
16. 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)
17. Olderog, E.-R., Apt, K.R.: Fairness in parallel programs: The transformational approach. ACM Trans. Program. Lang. Syst. 10(3), 420–455 (1988)
18. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
19. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS (2004)
20. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for Java bytecode based on path-length. ACM Trans. Program. Lang. Syst. (2010)