Proving Thread Termination

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
University of Freiburg
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
EPFL and MPI
rybal@mpi-sb.mpg.de

Abstract

Concurrent programs are often designed such that certain functions executing within critical threads must terminate. Examples of such cases can be found in operating systems, web servers, e-mail clients, etc. Unfortunately, no known automatic program termination prover supports a practical method of proving the termination of threads. In this paper we describe such a procedure. The procedure's scalability is achieved through the use of environment models that abstract away the surrounding threads. The procedure's accuracy is due to a novel method of incrementally constructing environment abstractions. Our method finds the conditions that a thread requires of its environment in order to establish termination by looking at the conditions necessary to prove that certain paths through the thread represent well-founded relations if executed *in isolation of the other threads*. The paper gives a description of experimental results using an implementation of our procedure on Windows device drivers, and a description of a previously unknown bug found with the tool.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering—Program Verification; D.4.5 [*Software*]: Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Concurrency, Formal verification, Model checking, Program verification, Termination, Threads

1. Introduction

Concurrent event-driven systems (*e.g.* operating systems, web servers, mail servers, database engines) usually execute event-handling routines in independent threads that communicate through shared-memory. It is often the case that the system's reliability and usability relies on termination guarantees from code executed in these threads. Device drivers, for example, provide event-handling dispatch routines that are allowed by the operating system to *temporarily* take over the execution context of the thread in which the event occurred. Failing to terminate while handling these events is considered as a violation of correctness. The complication for a programmer trying to ensure termination of a dispatch routine is that other dispatch routines from the same device driver may likely be concurrently modifying shared data and, thus, perhaps inadvertently cause the loop to diverge. The code in Figure 1, for example, is a demonstration of this problem. This loop, which comes from a keyboard device driver, could diverge if other threads from the same driver begin adding elements into the queue without first acquiring the spinlock `&Ext->SpinLock`. This scenario would render the keyboard and machine useless.

```
KeAcquireSpinLock(&Ext->SpinLock, &irql);

do {
  irp = DequeueReadByFileObject(Ext, FileObject);
  if (irp) {
    irp->IoStatus.Status = STATUS_CANCELLED;
    irp->IoStatus.Information = 0;

    InsertTailList (&listHead,LinkPtr(irp));
  }
} while (irp != NULL);

KeReleaseSpinLock(&Ext->SpinLock, irql);
```

Figure 1. Code fragment from a keyboard device driver whose termination partially depends on the correct behavior of other threads from the driver.

Unfortunately, today's automatic program termination provers are designed only to support sequential programs. Note that we cannot prove the termination of a thread by simply applying a sequential program termination prover individually to the thread: a sound proof of termination must consider the possible interactions with the other threads. Furthermore, merely encoding all possible interleavings between threads as a single sequential program does not lead to a scalable solution using today's termination provers.

What is needed is a practical, automatic, accurate and scalable termination-proof technique that supports threads. In an effort to achieve this goal, in this paper we adapt thread-modular verification techniques based on environment abstractions (*e.g.* [15, 26, 30, 33]) for the application of proving thread termination.

The challenge in this approach is to find a sound abstraction of the environment that is accurate enough to prove the property of interest. The technical contribution of this paper is a novel method of constructing sound environment abstractions in the context of thread-modular termination proofs. Our method finds the conditions that a thread requires of its environment in order to establish termination by looking at the conditions necessary to prove that certain paths through the thread represent well-founded relations if executed *in isolation of the other threads*. If such a proof exists, the method extracts information from this proof in order to incrementally strengthen the environment abstraction. The method also uses failed attempts to prove the soundness of the environment abstraction in order to incrementally weaken it.

To demonstrate the practical utility of the proposed method we have applied it to proving that Windows device driver dispatch routines do not diverge when executed in concurrent setting. These ex-

PLDI'07 June 11–13, 2007, San Diego, California, USA.

```
lock(lck);
while(x>0) {
  InterlockedDecrement(&x);
}
unlock(lck);
```

T_1

```
while(nondet()) {
  InterlockedDecrement(&x);
}
```

T_2

```
while(nondet()) {
  y = y + 1;
  lock(lck);
  x = nondet();
  unlock(lck);
}
```

T_3

Figure 2. Example concurrent program $\mathcal{P} = T_1||T_2||T_3$. The binary relation $\mathcal{A} \triangleq \mathtt{lck} \neq 1 \vee \mathtt{x}' \leq \mathtt{x}$ is an environment model (*a.k.a.* agreement) sufficient to show thread-termination of T_1. `InterlockedDecrement(&x)` is a Windows kernel-level API that implements `x:=x-1` atomically. Note that, without `InterlockedDecrement(&x)`, the instruction `x=x-1` would be treated non-atomically as two separate instructions (*e.g.* `t=x-1;x=t;`). The function `nondet()` is used to represent non-deterministic choice. We note that non-termination of the threads T_2 and T_3 does not affect the thread-termination property of the thread T_1.

periments represent the first known application of automatic thread-termination proofs for industrial software.

Related work. All automatic tools known to be successful at proving properties of concurrent programs employ some form of reduction or elimination of the number of interleavings and channels of interaction considered during their search for a proof. Examples include [13, 16, 24, 25, 27, 29–31, 34, 35]. In the case of tools that attempt to eliminate all interleavings (*e.g.* thread-modular tools [26,30]), the burden of performance and accuracy largely falls to the techniques used to find the environment abstractions that facilitate the use of only sequential program analysis tools. The distinguishing characteristic of our approach in comparison to previous thread-modular techniques is the novel method described for iteratively strengthening environment models for the application of thread-modular termination analysis.

A variety of program termination provers have been reported in the literature (*e.g.* for imperative programs see [2,5,8,9,17–19,21, 22,40]). Our work differs from these previous works in that we are the first to describe a practical method for proving termination of threads—to date all of the existing work on automatic termination analysis has focused on sequential programs.

In order to build a working solution to the thread-termination problem we have built upon previously reported techniques. We use existing tools for sequential program termination analysis. We also use sequential safety provers when checking the soundness of candidate abstract environment models. We use a well-known method for computing thread-local program invariants (*e.g.* [32]). We also use a technique of computing relational meanings for instructions via single-step symbolic simulation with fresh variables (as seen in *e.g.* [6,22,36,39]).

2. Example

Before providing a more formal description we begin with an example which shows the key aspects of the algorithm. Figure 2 contains a concurrent program $\mathcal{P}$ which is formed by the composition of three threads: T_1, T_2, and T_3. Imagine that we would like to prove the termination of T_1—meaning that no computation of the program $\mathcal{P}$ contains infinitely many T_1-steps. Notice that $\mathcal{P}$ itself does not guarantee termination.

Our algorithm is in search for an environment model, call it $\mathcal{A}$, that over-approximates the behavior of the threads T_2 and T_3 while being precise enough to facilitate the proof of T_1's termination. $\mathcal{A}$ is a binary relation over the states of the program $\mathcal{P}$ expressed using only primed and unprimed versions of $\mathcal{P}$'s shared variables (which are `x`, `x'`, `lck`, and `lck'` for our program). We use the primed variables, such as `x'` for example, to represent values of program variables after taking a transition.

In this work we call $\mathcal{A}$ an *agreement*, as our algorithm implements a form of two-way negotiation which involves both strengthening and weakening. The search for an $\mathcal{A}$ uses feedback both from the thread that we are trying to prove terminating and the threads in the environment. During its execution the algorithm finds a series of *draft agreements*, $\mathcal{A}_1$, $\mathcal{A}_2$, etc. The algorithm attempts to compute a fixpoint on these draft agreements. Both the draft agreements and the final agreement are represented in conjunctive normal form. Strengthenings are implemented as additional conjuncts, and weakenings as additional disjunctions within a single conjunct.

We begin with the weakest possible draft agreement

$$\mathcal{A}_1 \triangleq \textbf{true} \ ,$$

which imposes no constraints on the change of the shared variables by the threads T_2 and T_3. Clearly T_2 and T_3 meet this agreement.

Our algorithm then attempts to prove thread-termination of T_1 not in the original concurrent setting of $\mathcal{P}$, but instead in a concurrent setting represented by $\mathcal{A}_1$. This amounts to considering computations that arise from alternating execution steps of T_1 with non-deterministic updates that respect the agreement. In our case of the weakest possible agreement $\mathcal{A}_1 \triangleq \textbf{true}$, we must assume that the updates can change the value of `x` arbitrarily and the value of `lck` in any way that obeys the locking discipline assumed to hold of the program.

In the first iteration, the algorithm finds a counterexample to the termination of T_1 in the concurrent setting with the agreement $\mathcal{A}_1$. The counterexample is a sequence of instructions through T_1 and together with instructions representing the interleaving with $\mathcal{A}_1$. In this case the counterexample contains the following statements, where the statement `x=nondet()` models the effect of interleaving with the agreement $\mathcal{A}_1$:

```
x=x-1;
x=nondet();
assume(x>0);
```

Notice that this instruction sequence is not well-founded (*i.e.* the sequence can be repeated forever without reaching an inconsistent state). However, it is a spurious counterexample, as it does not represent an execution allowed in $\mathcal{P}$.

The algorithm then uses this path to strengthen the agreement. Strengthening here means that we add a conjunction to the agreement such that the found counterexample is eliminated in the next iteration of the algorithm. The relation

$$\mathcal{A}_2 \triangleq \textbf{true} \wedge \mathtt{x}' \leq \mathtt{x}$$

becomes the new agreement. The strengthening, $\mathtt{x}' \leq \mathtt{x}$, is inferred from a ranking relation for the restriction of the counterexample to the steps taken only by the thread T_1 (*i.e.* from the proof that the sequence of statements `x=x-1; assume(x>0);` is not a valid counterexample to termination).

In the second iteration, we succeed in proving termination of T_1 in the concurrent setting with the agreement. We must then check whether the other threads in the program $\mathcal{P}$, *i.e.* the threads T_2 and T_3, respect the draft agreement $\mathcal{A}_2$. In order to show that the thread T_2 respects the agreement $\mathcal{A}_2$, we must prove that `InterlockedDecrement(&x)`, which implements the decrement `x=x-1` atomically, does not increase the value of `x`. This check succeeds. We then attempt to prove that T_3 respects $\mathcal{A}_2$ by checking that the non-deterministic assignment `x=nondet()` respects $\mathcal{A}_2$. This check fails—meaning that the agreement $\mathcal{A}_2 \triangleq \textbf{true} \wedge \mathtt{x}' \leq \mathtt{x}$ is too strong to be satisfied by the thread T_3. In response to this failed check we attempt to weaken $\mathcal{A}_2$.

To weaken $\mathcal{A}_2$ we take the locking into account: The statement `x=nondet()`, which failed the check, is protected by a lock `lck`. Thus we weaken the agreement with a side-condition that rules out this case:

$$\mathcal{A}_3 \triangleq \textbf{true} \wedge (\mathtt{lck} \neq 1 \vee \mathtt{x}' \leq \mathtt{x}) \ .$$

The range of `lck` is over the thread identifiers $\{1, 2, 3\}$. Informally stated, $\mathcal{A}_3$ states that the environment should "ensure that it never increases the value of `x` in the case that thread T_1 holds the lock `lck`".

We now restart the algorithm with the weakened draft agreement $\mathcal{A}_3$. We can still prove the termination of T_1 in the context of the weaker agreement $\mathcal{A}_3$. Furthermore, we can now prove that both T_2 and T_3 respect $\mathcal{A}_3$. Thus, we have found a fixpoint and proved that T_1 is thread-terminating in the concurrent program $\mathcal{P}$ using the final agreement

$$\mathcal{A} \triangleq \mathcal{A}_3 \triangleq \textbf{true} \wedge (\mathtt{lck} \neq 1 \vee \mathtt{x}' \leq \mathtt{x}) \ .$$

Practical reasons for success. The strategy proposed in this section uses a collection of tools as sub-procedures which approximate solutions to undecidable problems (*e.g.* both safety and termination for infinite-state systems)—thus we cannot guarantee that the approach will always terminate, nor can we guarantee that counterexamples found with the approach will not be spurious. However, as we will see in Section 7, in practice our preliminary implementation performs well. The following notes provide some explanation as to why:

- Many loops used in industrial programs use termination conditions that only depend on thread-local variables. As we will see in Section 7, more than half of the loops from Windows device drivers are provably thread-terminating with $\mathcal{A}_1 \triangleq \textbf{true}$. In the remaining cases only a few rounds of strengthening and weakening are typically required. This is due to the fact that programmers typically use defensive techniques when writing loops, *e.g.* they typically do not depend on complex whole-program invariants to establish termination. It is for this reason that our strategy of considering program paths in isolation works: the reason that a path does not represent a divergent execution will likely not involve much reasoning about the surrounding environment.
- Our definition of thread-termination does not rule out deadlock, as deadlock is a safety property. That is, we are proving that T_1 eventually stops executing, not that T_1 eventually reaches a call to `exit` or `return`. This separation of concerns allows us to use methods that would otherwise be precluded.
- It is known that both termination and safety checking can be made practical for some classes of industrial sequential software (*e.g.* [3, 7, 21, 28]). In our setting we are applying this same class of tools on sequential programs that encode the interleaving of only two threads (as opposed to 3, 4, etc), one of which is a representation of a typically very simple artifact (the agreement $\mathcal{A}$). Thus, in practice, the sequential program analysis tools still perform adequately on these programs.
- Sequential program termination provers typically do not *prove* the validity of the counterexamples that they return. That is, these tools typically are attempting only to *prove termination* and not *prove non-termination*. In theory this could cause our analysis to loop forever as it tries to strengthen/weaken agreements based on counterexamples that are not only spurious in the concurrent setting but also the sequential setting. In practice, however, this is not a problem. If the termination prover supports linear arithmetic, bit-vectors, and heaps then the number of spurious counterexamples is quite low.
- The agreements necessary to prove thread termination typically need only express conditions on the directions in which the values of shared variables may change, *i.e.* the direction of variance of a shared variable rather than a unary invariance relationship. Aside from conditions on lock variables our method never introduces unary state assertions, *e.g.* $\mathtt{y} > 0$, but rather always uses binary variance assertions, *e.g.* $\mathtt{x}' \leq \mathtt{x}$. It is for this reason that we can easily prove that threads in the environment respect the agreement $\mathcal{A}$. In the case that $\mathcal{A} \triangleq \mathtt{y} > 0$, stronger methods would be required to prove that T_3 from Figure 2 respects $\mathcal{A}$.

3. Formal definitions

The previous section informally introduced our thread-modular algorithm for proving thread termination. In this section we make the algorithm more precise.

3.1 Preliminaries

We assume a fixed program $\mathcal{P}$ composed from N concurrent threads $T_1, \ldots, T_{\mathsf{N}}$. Let G be the set of global states defined as valuations of shared variables. Let L_i be the local states of thread T_i defined by valuations of local variables of thread T_i. A program state s consists of a global state g together with local states $l_1, \ldots, l_{\mathsf{N}}$ of the threads, *i.e.* $s = (g, l_1, \ldots, l_{\mathsf{N}})$. Let Σ be the set of all states of the program $\mathcal{P}$, *i.e.* $\Sigma = G \times L_1 \times \cdots \times L_{\mathsf{N}}$.

When making a transition from a state s, we assume that in practice each thread T_i can only modify the global state g and its local state l_i; all other local states l_j, where $j \neq i$, remain the same. Formally, the transition relation R_i of the thread T_i is a subset of $\Sigma \times \Sigma$ such that for each $((g, l_1, \ldots, l_{\mathsf{N}}), (g', l'_1, \ldots, l'_{\mathsf{N}})) \in R_i$ we have that for all $j \neq i$, $l_j = l'_j$ holds. Moreover, without loss of generality, we assume that $l_i \neq l'_i$. For example we can assume that each thread has a local program counter which changes its value if and only if the corresponding thread makes a step. This assumption is for Definition 1.

The transition relation R of the program $\mathcal{P}$ is the disjoint union of the transition relations of the threads, $R = R_1 \uplus \cdots \uplus R_{\mathsf{N}}$. Non-atomic reads and writes can be modelled in this framework by splitting single statements into composite statements with temporary variables (*e.g.* `x=x+1` can be re-written as `t=x; x=t+1;`).

Let $I \subseteq \Sigma$ be the set of initial states of the program $\mathcal{P}$. We define a computation σ of the program $\mathcal{P}$ to be a (possibly infinite) sequence of states $\sigma = s_1, s_2, \ldots$ such that s_1 is an initial state, and each pair of consecutive states (s, s') is in R. We say that a state is reachable if it appears in some computation, and write $\mathsf{Reach}(\mathcal{P})$ for the set of all reachable states of the program $\mathcal{P}$.

We assume a finite set of lock-variables, Locks. Assume that we have a lock $lck \in \mathsf{Locks}$. Intuitively, if the value of the lock lck in the state s is 3 then this means that the thread T_3 (and no other thread) holds the lock lck in the state s. If its value is 0 then no thread holds the lock.

We assume that threads satisfy the following locking discipline: A thread can acquire a lock only for itself, and it can release a lock only when it owns it. Thus, we are not currently considering mutexes. Formally, for each thread T_i and lock variable lck we assume $R_i \subseteq \mathsf{D}_i^A$ and $R_i \subseteq \mathsf{D}_i^B$, where

$$\mathsf{D}_i^A \triangleq \{(s, s') \mid s(lck) = 0 \wedge s'(lck) \neq 0 \rightarrow s'(lck) = i\}$$

$$\mathsf{D}_i^B \triangleq \{(s, s') \mid s(lck) \neq i \wedge s(lck) \neq 0 \rightarrow s'(lck) = s(lck)\}$$

Later in the paper we will use D_i^A and D_i^B as formulae over variables rather than sets of states, *e.g.* “$lck = 0 \wedge lck' \neq 0 \rightarrow lck' = i$”.

3.2 Thread termination

Assume that T_1 is the thread whose termination behavior is under investigation.

DEFINITION 1 (Thread termination). T_1 is *thread terminating* if in each computation of the program $\mathcal{P}$ it makes only finitely many steps. Formally, for every computation $\sigma = s_1, s_2, \dots$ of the program $\mathcal{P}$ there exists a position $k \geq 1$ such that there are no T_1 steps after k, *i.e.* if $(s_i, s_{i+1}) \in R_1$ then $i \leq k$.

DEFINITION 2 (Agreement $\mathcal{A}$). An *agreement* $\mathcal{A}$ is a binary relation over states that expresses a constraint on the change of global states, and leaves the change of local states unconstrained. Formally, for each pair of states $((g, \dots), (g', \dots)) \in \mathcal{A}$ and for each $i \in \{1, \dots, \mathsf{N}\}$ we have

$$\{(l_i, l_i') \mid ((g, \dots, l_i, \dots), (g', \dots, l_i', \dots)) \in \mathcal{A}\} = L_i \times L_i \;.$$

We define the relational composition of two relations A and B over states in the usual way, *i.e.* $A \circ B = \{(s, s'') \mid \exists s'.\ (s, s') \in A \wedge (s', s'') \in B\}$. Let A^* denote the reflexive transitive closure of A. For each $i \in \{1, \dots, \mathsf{N}\}$ we define a restricted identity relation id_{L_i} over states (which constrains only the local states of thread T_i):

$$\begin{aligned}\mathsf{id}_{L_i} = \{\ &((g, \dots, l_i, \dots), (g', \dots, l_i, \dots)) \mid \\ &(g, \dots, l_i, \dots) \in \Sigma \text{ and } (g', \dots, l_i', \dots) \in \Sigma\} \;.\end{aligned}$$

Ideally, we would like to use the agreement $\mathcal{A}$ to break the proof of thread termination for T_1 into the checking that i) T_1’s transition relation—when combined with $\mathcal{A}$ and restricted to states $\mathsf{Reach}(\mathcal{P}) \times \mathsf{Reach}(\mathcal{P})$—is well-founded, and ii) that the transition relations of the remaining threads respect $\mathcal{A}$ when restricted to $\mathsf{Reach}(\mathcal{P}) \times \mathsf{Reach}(\mathcal{P})$. However, the difficulty is that the restriction to $\mathsf{Reach}(\mathcal{P}) \times \mathsf{Reach}(\mathcal{P})$ requires us to consider the set of all possible interleavings between the various threads. For this reason, following existing techniques for thread-modular verification, we compute a *thread-local invariant* S_i for each thread T_i.

A thread-local invariant S_i approximates $\mathsf{Reach}(\mathcal{P})$ by constraining only the local part l_i of states of the thread T_i. This means that the global part of the state and the local parts of the other threads remain unrestricted.

DEFINITION 3 (Thread-local invariant S_i). A set of states S_i is a *thread-local invariant* for the thread T_i in the concurrent program $\mathcal{P}$ if it only constrains the part of the state that is local to the thread T_i, contains the initial states I, and is closed under the image operation of the transition relation R_i of the thread T_i that is applied in composition with the non-deterministic assignment to the global variables. Formally, S_i needs to satisfy the conditions:

$$\begin{aligned}\{g \mid (g, \dots) \in S_i\} &= G \\ \{l_j \mid (\dots, l_j, \dots) \in S_i\} &= L_j \qquad \text{for each } j \in \{1, \dots, \mathsf{N}\} \setminus \{i\} \\ I &\subseteq S_i \\ [(S_i \times \Sigma) \cap \mathsf{id}_{L_i}] \circ R_i &\subseteq \Sigma \times S_i\end{aligned}$$

T_1 is *thread terminating* if there exists an agreement $\mathcal{A}$ and thread-local invariants $S_2, \dots, S_\mathsf{N}$ such that

1. The sequential program $\mathsf{SeqComp}(T_1, \mathcal{A})$ terminates,
2. $[R_i \cap (S_i \times S_i)] \subseteq \mathcal{A}$ for each $i \in \{2, \dots, \mathsf{N}\}$.

Figure 3. Thread-termination proof rule.

We use thread-local invariants to restrict the domain and range of R_i when checking it with respect to the agreement $\mathcal{A}$. A proof rule making this precise is given in Figure 3. In the rule we use the sequential program $\mathsf{SeqComp}(T_1, \mathcal{A})$ defined below (intuitively, we abstract the interleaving of thread T_1 with the other threads by composing each of its statements with any number of non-deterministic, but $\mathcal{A}$-conform assignments to the global variables).

DEFINITION 4 ($\mathsf{SeqComp}$). Let $\mathsf{SeqComp}(T_1, \mathcal{A})$ be a sequential program with the set of initial states I_1 and the transition relation

$$[\mathcal{A} \cap \mathsf{id}_{L_1}]^* \circ R_1 \;.$$

Soundness of the proof rule in Figure 3 relies on the following lemma relating thread-local invariants and premises of the proof rule.

LEMMA 1. *A thread-local invariant S_i, where $i \in \{2, \dots, \mathsf{N}\}$, approximates the set of reachable states of the program $\mathcal{P}$,* i.e. $\mathsf{Reach}(\mathcal{P}) \subseteq S_i$.

PROOF We show that every reachable state s in $\mathsf{Reach}(\mathcal{P})$ is contained in S_i by induction over the shortest number of transitions n which is required to reach s. For $n = 0$ we have that s is in I, and hence in S_i. For the induction step, we assume that s is reachable after n transitions, and hence by the induction hypothesis it is in S_i We prove that its successor s' is in S_i. Let T_j be the thread that makes the transition from s to s', *i.e.* $(s, s') \in R_j$. Now we consider the case when $i = j$. We have $(s, s) \in [(S_i \times \Sigma) \cap \mathsf{id}_{L_i}]$, from which follows that $(s, s') \in \Sigma \times S_i$. If $i \neq j$ then the l_i-part of s does not change during transition. Since S_i restricts only T_i-local part of the state, we have $s' \in S_i$. □

THEOREM 1. *The proof rule in Figure 3 is sound. Existence of an agreement $\mathcal{A}$ and a set of thread-local invariants $S_2, \dots, S_\mathsf{N}$, which satisfy the premises of the proof rule, implies the thread-termination property of the thread T_1.*

PROOF Assume that the conditions of the proof rule hold for an agreement $\mathcal{A}$ and thread-local invariants $S_2, \dots, S_\mathsf{N}$, but T_1 is *not* thread terminating. We shall derive a contradiction to the first condition, *i.e.* we show that the sequential program $\mathsf{SeqComp}(T_1, \mathcal{A})$ does not terminate.

Let $\sigma = s_1, s_2, \dots$ be an infinite computation of $\mathcal{P}$ that violates the thread termination property for T_1. Then, σ contains infinitely many transitions of T_1, which are possibly interleaved with transitions of other threads. We show that σ is a computation of $\mathsf{SeqComp}(T_1, \mathcal{A})$. We need to prove that each transition (s, s') in σ that is not taken by the thread T_1 satisfies the agreement $\mathcal{A}$, as we have $(s, s') \in \mathsf{id}_{L_i}$. Let $(s, s') \in R_i$ for some $i \in \{2, \dots, \mathsf{N}\}$. From Lemma 1, we have that s and s' are in S_i. Hence, from the second condition of the proof rule we have $(s, s') \in \mathcal{A}$. □

3.3 Algorithm

Our algorithm for proving thread-termination is defined over four sub-procedures: a *termination checker* (for sequential programs), a *strengthening procedure*, a *guarantee checker*, and a *weakening procedure*. See Figure 4 for the algorithm.

The termination checker is used in Figure 4 during attempts to prove the termination of sequential programs constructed from T_1

```
input
 concurrent program $\mathcal{P} = T_1 || \ldots || T_{\mathsf{N}}$
begin
 $\mathcal{A}$ := true
 repeat
     if $\mathsf{SeqComp}(T_1, \mathcal{A})$ terminates then
         return "$T_1$ is thread terminating"
     else
         $\pi$ := lasso counterexample in $T_1$
         if strengthening of $\mathcal{A}$ is not possible using $\pi$ then
             return "$T_1$-termination proof failed"
         else
             $\mathcal{A}$ := strengthened agreement found with $\pi$
         foreach $i \in \{2, \ldots, \mathsf{N}\}$ do
             if $T_i$ fails to guarantee $\mathcal{A}$ then weaken $\mathcal{A}$
         done
      done
 done
end.
```

Figure 4. Algorithm for checking thread-termination using termination agreements.

and $\mathcal{A}$, *i.e.* $\mathsf{SeqComp}(T_1, \mathcal{A})$. When this proof succeeds then we can argue that T_1 is thread terminating if $\mathcal{A}$ is a sound overapproximation of $T_2, \ldots T_{\mathsf{N}}$, *i.e.* the environment respects $\mathcal{A}$. If the sequential termination check fails then we expect that the termination prover returns a potential counterexample, which is a lasso path $\pi^{\mathcal{A}}$ through $\mathsf{SeqComp}(T_1, \mathcal{A})$. We say that $\pi^{\mathcal{A}}$ is defined to be a lasso path if $\pi^{\mathcal{A}} = \pi^{\mathcal{A},s}.(\pi^{\mathcal{A},c})^{\omega}$ and both $\pi^{\mathcal{A},s}$ and $\pi^{\mathcal{A},c}$ are finite sequences (the s is short for "stem", c is short for "cycle"). As is done in [21], we can actually think of $\pi^{\mathcal{A}}$ as a sequential program (independent of $\mathcal{P}$ and its threads) that represents executions of the form $\pi^{\mathcal{A},s}.(\pi^{\mathcal{A},c})^{\omega}$. We can even use other termination provers to try and prove that these "representative programs" are terminating, *i.e.* that the paths are spurious counterexamples to termination. Furthermore, we can remove the $\mathcal{A}$-steps from $\pi^{\mathcal{A}}$. Let this version of $\pi^{\mathcal{A}}$ be π. The path π can also be viewed as a program representing all executions in T_1 of the form $\pi^s.(\pi^c)^{\omega}$ where, again, π^s and π^c are finite sequences.

The strengthening procedure in Figure 4 attempts to prove the cyclic component of the sequential program that π represents is terminating (*i.e.* that the relation that it represents is well-founded) in order to mine information about the assumptions under which the thread T_1 terminates. If the strengthening procedure succeeds in proving well-foundedness of the cycle, then $\pi^{\mathcal{A}}$ may represent a spurious counterexample to T_1's thread-termination in $\mathcal{P}$. In this case the strengthening procedure strengthens the agreement $\mathcal{A}$ using the information mined from the proof of π^c's well-foundedness. Under the assumption that the environment satisfies the new agreement, the path π is not a counterexample to thread-termination in the concurrent context with the new agreement. A detailed description of this procedure and its connection to the sequential termination checker is contained in Section 5.

The guarantee checker in Figure 4 takes an agreement $\mathcal{A}$ and a thread T_i and attempts to ensure that T_i in its concurrent execution respects the agreement $\mathcal{A}$. If the guarantee checker fails then the the weakening procedure is used. The weakening procedure takes as input the agreement $\mathcal{A}$, the thread T_i, and an explanation for the failure of the guarantee check. The weakening procedure attempts to produce a new weaker agreement respected by the thread T_i. A detailed description of the checking and weakening mechanisms appear in Section 6.

4. Proving termination with agreements

Our goal in this section is to provide a method of encoding $\mathsf{SeqComp}(T_1, \mathcal{A})$ such that it can be passed to an existing sequential termination prover and such that the counterexamples produced by the termination prover can be examined in isolation of $\mathcal{A}$.

We assume that $\mathcal{A}$ is a formula expressed over the shared variables of the program, say V (including Locks), and their primed versions V' (including Locks'). For now we will assume that $\mathcal{A}$ is reflexive and transitive—later we will discuss an implementation-level detail that encodes the reflexive and transitive closure of $\mathcal{A}$.

To implement $\mathsf{SeqComp}(T_1, \mathcal{A})$ we define a new sequential program $\mathcal{S}$ with a control-flow graph that is identical to that of T_1, with the exception of additional instructions within the basic blocks. The construction of $\mathcal{S}$ is thus performed at the instruction-level on the instructions from the thread T_1.

DEFINITION 5 (Transformation implementing $\mathsf{SeqComp}(T_1, \mathcal{A})$). Let $\mathcal{S}$ be a program constructed from the thread T_1 such that $\mathcal{S}$ has the same control-flow graph as T_1 and a call to the procedure A_STAR( ) is placed in front of each statement. We define A_STAR( ) to be:

1. a series of assignment statements $v' := \texttt{nondet}()$; for each $v' \in \mathsf{V}'$ (including Locks'),
2. assume($\mathcal{A}$);
3. assume(D_j^A); for each $j \in \{2, \ldots, \mathsf{N}\}$ and for each lock variable lck,
4. assume(D_j^B); for each $j \in \{2, \ldots, \mathsf{N}\}$ and for each lock variable lck,
5. a series of assignment statements $v := v'$; for each $v' \in \mathsf{V}'$ (including Locks'),

Our transformation creates a program whose transition relation corresponds to the sequential composition of the transition relation of the thread T_1 and the relation defined by the agreement $\mathcal{A}$. It is achieved by having a statement that correspond to $\mathcal{A}$-respecting steps before each statement of the thread T_1.

Counterexamples. The counterexample produced by the sequential program termination prover will include instructions inside of A_STAR. Thus, this counterexample produced from $\mathcal{S}$ is the $\pi^{\mathcal{A}}$ mentioned in Section 3. In order to construct the T_1-centric counterexample π we need only remove the instructions from A_STAR from $\pi^{\mathcal{A}}$.

4.1 Examples

Agreements without locks. We consider the T_1 in Figure 2, and assume the agreement $\mathcal{A} \triangleq \texttt{x}' \leq \texttt{x}$. Informally this means

> "The other threads executing concurrently will only make x smaller if they modify x at all."

An application of the transformation from Definition 5 on the thread T_1 produces the program shown in Figure 5.

TERMINATOR, for example, is able to prove the termination of this sequential program. Furthermore, the termination proof of this program implies that T_1 terminates when executed in an environment which agrees to A.

Agreements with locks. We can also support agreements with predicates over locks, such as

$$\mathcal{A} \triangleq [\texttt{lck} \neq 1 \vee \texttt{x}' \leq \texttt{x}] \,.$$

The intention of $\mathcal{A}$ is almost the same as before with the extra side condition that the guarantee need only be provided by the other threads when thread 1 holds the lock lck. In case that another thread does have the lock lck, then $\texttt{lck} \neq 1$ and thus, nothing is promised.

```
void A_STAR()
{
    lck’ = nondet();
    x’ = nondet();
    assume(!(lck==0 && lck’!=0) || lck’>1);
    assume( lck!=1 || lck’==l);
    assume(x’<=x);
    lck = lck’;
    x = x’;
}

void 𝒮()
{
     A_STAR();
     lock(lck)
     A_STAR();
     while(x>0) {
         A_STAR();
         InterlockedDecrement(&x);
         A_STAR();
     }
     A_STAR();
     unlock(lck);
}
```

Figure 5. Result of transformation in Definition 5 on T_1 from Figure 2 and $\mathcal{A} = \mathtt{x}' \leq \mathtt{x}$.

```
void A_STAR()
{
    lck’ = nondet();
    x’ = nondet();
    assume(!(lck==0 && lck’!=0) || lck’>1);
    assume( lck!=1 || lck’==l);
    assume(x’<=x || lck!=1);
    lck = lck’;
    x = x’;
}

void 𝒮()
{
     A_STAR();
     lock(lck)
     A_STAR();
     while(x>0) {
         A_STAR();
         InterlockedDecrement(&x);
         A_STAR();
     }
     A_STAR();
     unlock(lck);
}
```

Figure 6. $\mathcal{S}$ constructed from $\mathcal{A} \triangleq [\mathtt{lck} \neq 1 \vee \mathtt{x}' \leq \mathtt{x}]$ and T_1 from Figure 2.

We consider again the thread T_1 from Figure 2, in which the instruction `lock(lck)` is placed just before the start of the loop. Using this example together with our new $\mathcal{A}$ will give us a sequential program in which the definition of `A_STAR` is contained in Figure 6. The available program termination provers can prove the termination of this program also. However, without the call to `lock(lck)` the termination proof would fail.

4.2 Supporting non-transitive agreements

In the case that the agreement $\mathcal{A}$ is not transitive, we can encode its reflexive and transitive closure in our translation by placing the following code-fragment instead of `A_STAR()` in front of each statement in the thread T_1:

```
while(nondet()) {
    IgnoreCutpoint();
    A_STAR();
}
```

We assume that the use of `IgnoreCutpoint()` causes the program termination prover to ignore failures of program termination in which the infinite subsequence of the non-termination execution remains strictly within the loop enclosing calls to `IgnoreCutpoint()`. In practice (where termination provers are based on the identification of cutpoints in the program's control-flow graph) this is easy to implement.

5. Strengthening agreements

In this section we describe a method of strengthening the termination agreement $\mathcal{A}$ with additional constraints. This has the effect of placing additional constraints on the environment, thus giving the thread of interest more guarantees about the direction of the variance of the variables that it is reading and updating.

We assume the situation where the sequential termination prover has found a counterexample path $\pi^{\mathcal{A}}$ in the sequential program $\mathcal{S}$. This means that there exists a sequence of statements only from the thread T_1. Let π be the subsequence of $\pi^{\mathcal{A}}$ that consists of the statements from the thread T_1. Recent termination provers can produce such counterexamples and represent them in form of lasso paths (as described in [21]). Hence, we assume that π consists of two parts, π^s and π^c. For simplicity of exposition, we only use the cycle part of the lasso for refining the agreement. In order to increase precision, the stem part can be taken into account following an algorithm in [21].

The strengthening algorithm analyzes whether the non-termination is caused by the interleaving with $\mathcal{A}$. This analysis amounts to the computation of a ranking function for the sequence π^c (which does not contain any statements from $\mathcal{A}$). If such a ranking function exists then we say that the counterexample is potentially spurious. We say *potentially* spurious because $\mathcal{A}$ might truly represent a possible behavior of the environment.

In the case that π^c is well-founded, the strengthening algorithm suggests a strengthening $\mathcal{A}_\delta$ of the termination agreement that can be used for pruning the counterexample on the next iteration of the algorithm. We thus use $\mathcal{A}_\delta$ to strengthen $\mathcal{A}$:

$$\mathcal{A} := \mathcal{A} \wedge \mathcal{A}_\delta$$

$\mathcal{A}_\delta$ needs to satisfy the following properties.

- Any interleaving of $\mathcal{A}_\delta$ with the statements from π^c is a well-founded relation, i.e. the sequence below is well-founded.

  $$\mathcal{A}_\delta^* \pi_1^c \mathcal{A}_\delta^* \dots \mathcal{A}_\delta^* \pi_n^c \mathcal{A}_\delta^*$$

  Here, $\mathcal{A}_\delta^*$ represents an arbitrary sequence of $\mathcal{A}_\delta$ elements. This property guaranties that our thread-termination algorithm makes progress at each iteration—each counterexample appears only once. Section 6 describes an algorithm that uses locking information to weaken agreements, while maintaining the progress property.
- $\mathcal{A}_\delta$ must be a reflexive relation, and hence admit void environments not modifying shared variables.

We argue that the binary relation that captures the non-increase of the ranking function for π^c, together with the identity relation provide a good candidate for a stronger termination agreement. Let $r(x)$ be an expression over the L_1-variables that determines the ranking function computed for the sequence π^c. We define

$$\mathcal{A}_\delta \;\equiv\; r(x') \leq r(x) \,.$$

We observe that $\mathcal{A}_\delta$ is transitive and reflexive as required by our algorithm. Additionally, $\mathcal{A}_\delta$ guarantees the elimination of this counterexample if all assertions that are used in the proof of well-foundedness of π^c are transitive, which is often observed in systems code.

LEMMA 2. *Let the sequential composition of statements* $\mathtt{stmt}_1$ *and* $\mathtt{stmt}_2$ *given over the variables x and x' be included in the 'strictly decreasing' relation $rx' \leq rx - 1$ induced by a linear ranking function rx. Then, for $\mathcal{A}_\delta = rx' \leq rx$ we have that the transition relation of the sequence of statements*

$$\mathtt{stmt}_1;\ \mathtt{x}' = \mathtt{nondet}();\ \mathtt{assume}(\mathcal{A}_\delta);\ \mathtt{x} = \mathtt{x}';\ \mathtt{stmt}_2;$$

is included in $rx' \leq rx - 1$ provided that the transition relations of the statements $\mathtt{stmt}_1$; *and* $\mathtt{stmt}_2$; *are transitive.*

PROOF Let the transition relations of the statements $\mathtt{stmt}_1$ and $\mathtt{stmt}_2$ be represented by the relations $(PP')\binom{x}{x'} \leq p$ and $(QQ')\binom{x}{x'} \leq q$, respectively. First, from their transitivity follows that $P = -P'$ and $Q = -Q'$. (The proof relies on the assumption that the rows of (PP') and (QQ') are linearly independent, which holds for transition relations of program statements.) Second, since their sequential composition is included in the 'strictly decreasing' relation, i.e.

$$\begin{pmatrix} P & P' & 0 \\ 0 & Q & Q' \end{pmatrix} \begin{pmatrix} x \\ x' \\ x'' \end{pmatrix} \leq \begin{pmatrix} p \\ q \end{pmatrix} \implies \begin{pmatrix} -r & 0 & r \end{pmatrix} \begin{pmatrix} x \\ x' \\ x'' \end{pmatrix} \leq -1 \,,$$

we have that for some non-negative vectors λ and μ

$$\lambda P = -r \qquad \lambda P' = -\mu Q \qquad \mu Q' = r \,.$$

We consider the sequential composition of the statements interleaved with the agreement:

$$\begin{pmatrix} P & P' & 0 & 0 \\ 0 & -r & r & 0 \\ 0 & 0 & Q & Q' \end{pmatrix} \begin{pmatrix} x \\ x' \\ x'' \\ x''' \end{pmatrix} \leq \begin{pmatrix} p \\ 0 \\ q \end{pmatrix} \,.$$

We observe that the linear combination defined by the vector $(\lambda\ 1\ \mu)$ yields the desired implication $rx''' \leq rx - 1$. □

Now by induction over the length of the sequence π^c we can prove that the transition relation of the sequence $\mathcal{A}_\delta^* \pi_1^c \mathcal{A}_\delta^* \ldots \mathcal{A}_\delta^* \pi_n^c \mathcal{A}_\delta^*$ is included in $rx' \leq rx - 1$.

5.1 Example

Consider a program $\mathcal{S}$ representing $\mathsf{SeqComp}(T_1, \mathcal{A})$, where T_1 is drawn from Figure 2, and $\mathcal{A} = \textbf{true}$. See Figure 7. In this case a sequential program termination prover would produce counterexample including the cycle expressed by the line numbers of the traversed statements.

$$\pi^{\mathcal{A},c} = 3.A \to 3 \to 4.A \to 2 \to$$

```
A.1   void A_STAR(int i)
A.2   {
A.3       x' = nondet();
A.4       lck' = nondet();
A.5       assume(!(lck==0 && lck'!=0) || lck'>1);
A.6       assume( lck!=1 || lck'==lck);
A.7       assume(true);
A.8       x = x';
A.9       lck = lck';
A.10  }

0     void S() {
1.A       A_STAR();
1         lock(lck);
2.A       A_STAR();
2         while(x>0) {
3.A           A_STAR();
3             InterlockedDecrement(&x);
4.A           A_STAR();
4         }
5.A       A_STAR();
5         unlock(lck);
6     }
```

Figure 7. $\mathcal{S}$ constructed from $\mathcal{A} = \textbf{true}$ and T_1 from Figure 2.

When we consider the representation of $\pi^{\mathcal{A},s}$ in static single assignment form, we get the following relation, $[\![\pi^{\mathcal{A},c}]\!] =$

$$\begin{array}{ll}
\{\ ((\mathtt{x}_0, \mathtt{lck}_0), (\mathtt{x}_3, \mathtt{lck}_2)) & \\
\ \mid \mathtt{lck}_0 = 0 \wedge \mathtt{lck}_1 \neq 0 \Rightarrow \mathtt{lck}_1 > 1 & \text{// 3.A: creating } \mathtt{x}_1 \text{ and } \mathtt{lck}_1 \\
\wedge\ \mathtt{lck}_0 = 1 \Rightarrow \mathtt{lck}_1 = \mathtt{lck}_0 & \text{// 3.A:} \\
\wedge\ \mathtt{x}_2 = \mathtt{x}_1 - 1 & \text{// 3:} \\
\wedge\ \mathtt{lck}_1 = 0 \wedge \mathtt{lck}_2 \neq 0 \Rightarrow \mathtt{lck}_2 > 1 & \text{// 4.A: creating } \mathtt{x}_3 \text{ and } \mathtt{lck}_2 \\
\wedge\ \mathtt{lck}_1 = 1 \Rightarrow \mathtt{lck}_2 = \mathtt{lck}_1 & \text{// 4.A:} \\
\wedge\ \mathtt{x}_3 > 0 & \text{// 2:} \\
\} &
\end{array}$$

Because the $\mathcal{A}$-statements non-deterministically change the value of the variable $\mathtt{x}$, we can always find a value for $\mathtt{x}_3$ such that $\mathtt{x}_3 > 0$. Thus the relation that $\pi^{\mathcal{A},c}$ represents is not well-founded. However, if we consider only the statements in $\pi^{\mathcal{A},c}$ that come from T_1 (*i.e.* $\pi^c = 3 \to 2 \to$) we get a relation $[\![\pi^c]\!]$ that is provably well-founded.

$$[\![\pi^c]\!] = \{((\mathtt{x}_0, \mathtt{lck}_0), (\mathtt{x}_1, \mathtt{lck}_0)) \mid \mathtt{x}_1 = \mathtt{x}_0 - 1 \wedge \mathtt{x}_1 > 0\}$$

In this case tools based on rank function synthesis (*e.g.* RANKFINDER [38] or POLYRANK [9–12]) would produce a witness to the well-foundedness in the form of a ranking function $\mathtt{x}$. Every relation step decreases the value of the ranking function by at least one, and this decrease is bound from below by zero.

We are now in the situation where the counterexample found in $\mathcal{S}$ is either due to a real concurrency bug or an insufficient environment assumption. The observation that we make here is that the computation of the ranking function in the non-concurrent case can provide a useful hint for the construction of the environment assumption that the developer of the thread T_1 is expecting. Following Lemma 2 we consider the relation

$$\mathcal{A}_\delta = x' \leq x$$

for strengthening of the agreement. Note that a termination prover for sequential programs does not produce the same counterexample when supplied a program implementing $\mathsf{SeqComp}(T_1, \mathcal{A} \wedge \mathcal{A}_\delta)$, as the relation represented by the cycle $\pi^{\mathcal{A} \wedge \mathcal{A}_\delta, s}$ is well-founded,

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![\pi^{\mathcal{A}\wedge\mathcal{A}_\delta,c}]\!] =} \\
\quad \{\ ((\texttt{x}_0, \texttt{lck}_0), (\texttt{x}_3, \texttt{lck}_2)) & \\
\quad\ \mid \texttt{lck}_0 = 0 \wedge \texttt{lck}_1 \neq 0 \Rightarrow \texttt{lck}_1 > 1 & \text{// 3.A: creating } \texttt{x}_1 \text{ and } \texttt{lck}_1 \\
\quad \wedge\ \texttt{lck}_0 = 1 \Rightarrow \texttt{lck}_1 = \texttt{lck}_0 & \text{// 3.A:} \\
\quad \wedge\ \texttt{x}_1 \leq \texttt{x}_0 & \text{// 3.A} \\
\quad \wedge\ \texttt{x}_2 = \texttt{x}_1 - 1 & \text{// 3:} \\
\quad \wedge\ \texttt{lck}_1 = 0 \wedge \texttt{lck}_2 \neq 0 \Rightarrow \texttt{lck}_2 > 1 & \text{// 4.A: creating } \texttt{x}_3 \text{ and } \texttt{lck}_2 \\
\quad \wedge\ \texttt{lck}_1 = 1 \Rightarrow \texttt{lck}_2 = \texttt{lck}_1 & \text{// 4.A:} \\
\quad \wedge\ \texttt{x}_3 \leq \texttt{x}_2 & \text{// 4.A} \\
\quad \wedge\ \texttt{x}_3 > 0 & \text{// 2:} \\
\quad \} &
\end{array}
$$

6. Checking and weakening agreements

In this section, we describe a method for checking if the other threads in the program $\mathcal{P}$ respect the agreement $\mathcal{A}$. Our method relies on a *sequential* safety checker for the computation of thread-local invariants $S_2, \ldots, S_{\mathsf{N}}$. We also describe a method of weakening $\mathcal{A}$ in the case that the environment does not respect it.

We check that the threads $T_2, \ldots, T_{\mathsf{N}}$ respect $\mathcal{A}$ by proving the assertion validity for a set of sequential programs $\mathsf{AgChk}(T_2, \mathcal{A})$, ..., $\mathsf{AgChk}(T_{\mathsf{N}}, \mathcal{A})$ (defined below in Definition 6). Each program is constructed by adding additional statements into the control-flow graph of the thread, and using auxiliary variables ${}'\mathsf{V}$ that are copies of the shared variables V. These statements serve two goals. Firstly, they ensure that the set of reachable states of a program $\mathsf{AgChk}(T_i, \mathcal{A})$ corresponds to a thread-local invariant S_i for the thread T_i. Secondly, they check if the agreement $\mathcal{A}$ is satisfied. This check naturally takes the thread-local invariant S_i into account.

DEFINITION 6 ($\mathsf{AgChk}(T_i, \mathcal{A})$). Let $\mathcal{Q}_i = \mathsf{AgChk}(T_i, \mathcal{A})$ be a program constructed from the thread T_i, where $i \in \{2, \ldots, \mathsf{N}\}$ such that $\mathcal{Q}_i$ has the same control-flow graph as T_i and each statement `stmt;` is replaced by the following sequence of statements:

1. a series of assignment statements $v' := \texttt{nondet}()$; for each $v' \in \mathsf{V}'$,
2. `assume`$(\neg(lck = 0 \wedge lck' \neq 0) \vee lck' \neq i\ )$; for each lock variable lck,
3. `assume`$(lck \neq i \vee lck' = i)$; for each lock variable lck,
4. a series of assignment statements ${}'v := v$; for each ${}'v \in {}'\mathsf{V}$,
5. a series of assignment statements $v := v'$; for each $v \in \mathsf{V}$,
6. `stmt;` and
7. `assert`$(\mathcal{A}[{}'\mathsf{V}/\mathsf{V}][\mathsf{V}/\mathsf{V}'])$);

The output of this transformation, $\mathcal{Q}_i$ can be passed to a sequential safety checker. We observe that the set of reachable states of $\mathcal{Q}_i$ represent a thread-local invariant S_i that satisfies the conditions of the proof rule and is sufficiently strong to support the proof of agreement satisfaction. In practice, the accuracy of our method depends on determining which locks and relations between thread variables are held at each program location within the threads $\{2, \ldots \mathsf{N}\}$. This information can be represented as arithmetic program invariants and can thus be computed using existing analysis tools for sequential programs (*i.e.* [23, 37]) or by adding additional predicates over the lock variables before applying the property-driven safety checker. We assume that all threads in the program respect locking discipline D^A and D^B. That is: our algorithm never checks that the the lock-discipline is respected, though this can be easily assured via a static check that threads only use `lock` and `unlock` to modify the values of lock variables.

THEOREM 2. *If the program $\mathcal{Q}_i$ satisfies its assertions then the thread T_i respects agreement $\mathcal{A}$,* i.e.*, the condition 2 in the proof rule holds for the thread T_i.*

Weakening. If the agreement $\mathcal{A}$ is not a sound overapproximation of the environment threads then there will exist a thread T_i, where $i \in \{2, \ldots, \mathsf{N}\}$, such that an `assert` statement in the corresponding sequential program $\mathcal{Q}_i$ fails. Let k be the location of this `assert` in the body of $\mathcal{Q}_i$. Also assume that $\mathcal{A} = \mathcal{A}' \wedge \rho$, and that the conjunct ρ causes the assertion failure at the location k. In this case we weaken the agreement by replacing ρ in $\mathcal{A}$ with $\rho \vee lck \neq 1$ if we can prove that $\texttt{pc} = k \Rightarrow lck = i$ is a thread-local invariant of the thread T_i for some lock variable $lck \in \mathsf{Locks}$. If we cannot find such a lock, we report a case of potential thread-nontermination together with the last counterexample π^c examined.

We observe that such weakening algorithm preserves the progress property of the overall method, *i.e.* that no counterexample is discovered more than once. The strengthening phase eliminates the counterexample by restricting the agreement. If any weakening takes place, then it does not undo the strengthening since it is only applied when the thread holds a lock.

6.1 Example

Consider T_3 from Figure 2, and the agreement $\mathcal{A} = [\texttt{l} \neq 1 \vee \texttt{x}' \leq \texttt{x}]$. We show how our method checks that the thread T_3 respects the agreement $\mathcal{A}$ on example of the statement `x=nondet();`.

The transformation $\mathsf{AgChk}(T_3, \mathcal{A})$ produces the following code fragment for the statement above.

```
lck’ = nondet();
x’ = nondet();
assume(!(lck==0 && lck’!=0) || lck’!= 3);
assume(lck!=3 || lck’=3);
lck = lck’;
x = x’;
‘x = x;
‘lck = lck;
x = nondet();
assert( ‘lck!=1 || x <= ‘x );
```

By applying a program analysis on the sequential program $\mathsf{AgChk}(T_3, \mathcal{A})$, we can check that the assertion holds. The proof relies on the discovery of the fact that $\texttt{lck} = 3$ when the assertion is checked.

7. Experimental results

In order to evaluate the effectiveness of our thread-termination algorithm, we have constructed a preliminary implementation and applied it to the problem of proving thread-termination of dispatch routines from Windows device drivers. We found that the translation implementing $\mathsf{SeqComp}(T_1, \mathcal{A})$ produces programs that are unnaturally difficult for termination provers based on [21]. The termination proof techniques described in [4], however, are largely unaffected by the translation. The technique from [4], however, does not produce counterexamples, whereas [21] does. Thus, during these experiments we used an implementation of a sequential termination prover that combined the techniques from [4] and [21].

As described in [5], termination proofs for some[1] loops in Windows device drivers require an analysis that is capable of reasoning about the lengths of linked data-structures. Following [8], we have added integer length-variables to our model of queues and other data-structures accessed by kernel-level APIs by the device drivers. The Windows kernel “interlocked dequeue” operation, for example, is modelled to atomically decrement a counter stored in

[1] Perhaps 30% on average.

the queue data-structure. Our change was not made in code being proved terminating, but rather the model of the operating system that is used in conjunction with the device driver.

Note that locks in Windows device drivers are stored in the heap, and not as global variables. However, in all cases seen in our evaluation, the locks were stored in a single data-structure created during the driver's startup: thus we modified the device driver code used in our evaluation to use global lock variables. An additional complication is that, in some circumstances, Windows device drivers use locking mechanisms not supported in our current formulation (*e.g.* mutexes). These cases were were avoided in our evaluation.

Results. See Table 1 for the results of our experiments. Example 17 includes the code from Figure 1, and Example 1 includes the code from Figure 8. Each example in Table 1 represents a thread-termination proof for a single loop (*i.e.* cutpoint in the control-flow graph) within a dispatch routine. Dispatch routines usually contain between 2 to 30 loops each, thus a complete thread-termination proof for a whole dispatch routine will require more processing time (perhaps 2x, 30x, or in some cases more). The device drivers range in sizes from 1,000 to 30,000 lines of code, however the reachable code from a single dispatch routine (device drivers usually export up to 10) usually ranges from 300 to 10,000 of lines of code.

Table 1 demonstrates promise that our algorithm from Figure 4 can be made to be practical, automatic, accurate and scalable. The tool is completely automatic. Only two false negatives were reported, both due to inaccuracies in the underlying termination prover's treatment of bitvectors. Furthermore, the performance (while expensive) in many cases is not intractable. Table 1 also demonstrates that, at least in this domain, simple agreements suffice. We expect the same to be true for most instances of industrial software.

Comparison with existing tools. As mentioned in Section 1, until now no known termination prover has "natively" supported thread termination. Before now the sound option available was to apply a sequential termination prover on a program that represents an encoding of all of the interleavings of T_1 to T_N. In order to compare against our new algorithm we have tried this for the first three examples from Table 1: all three cases resulted in a timeout after 3 hours using TERMINATOR.

The other potential competitor to our technique is simply to run a sequential termination prover on the single thread in question, thus simply ignoring the unsoundness (due to bugs that can only be found in the concurrent setting). During the analysis that produced Table 1 we found 3 previously unknown bugs (Examples 1, 5, and 19). The bugs in Examples 1 and 19 indeed require support for concurrency in order to be found, meaning that in principle no other known program termination prover would be able to find these bugs. This assertion was verified experimentally for at least one sequential termination prover (*i.e.* TERMINATOR)—meaning that we applied TERMINATOR to the individual threads with bugs from Examples 1 and 19 and found that the sequential tool reported an unsound result due to the fact that it was ignoring the interleavings with the environment. As $\mathcal{A} = \textbf{true}$ in Example 5, however, this bug can be found simply with a sequential termination prover.

Notes on the bug from Example 1 of Table 1 (Figure 8). Figure 8 shows the loop from the first example in Table 1. This is not *all* of the code used in the example, just the body of the loop. All of the code reachable from this loop was considered during the thread-termination proof together with all of the code from all of the other threads that could be executing concurrently with it. This example comes from the 'I/O control' dispatch routine of a modem driver. During the execution of this dispatch routine, the code in the "read-request" dispatch routine (not displayed) could be executed many times concurrently. This "read-request" dispatch routine can, in cases, add elements to the queue that is being emptied in Figure 8. Thus, if an ongoing supply of read-requests arrive concurrently during the execution of the loop in Figure 8, the thread executing the "I/O control" dispatch routine may not terminate.

Example	Time	Result	Strengthenings	Weakenings
1	6192s	CEX	1	1
2	3235s	Pass	1	1
3	1366s	Pass	0	0
4	T/O	-	-	-
5	932s	CEX	0	0
6	833s	Pass	1	0
7	145s	False	0	0
8	T/O	-	-	-
9	2116s	False	0	0
10	T/O	-	-	-
11	521s	Pass	0	0
12	4799s	Pass	1	1
13	167s	Pass	0	0
14	T/O	-	-	-
15	6733s	Pass	2	2
16	99s	Pass	1	1
17	339s	Pass	1	1
18	8831s	Pass	2	1
19	1955s	CEX	1	1
20	4224s	Pass	1	1
21	54s	Pass	0	0
22	28s	Pass	0	0
23	T/O	-	-	-
24	139s	Pass	1	1
25	344s	Pass	0	0

Table 1. Results of experimental evaluation on device drivers. Each example represents a single loop in a dispatch routine for a device driver. "Strengthenings" indicates how many times the strengthening procedure was called, and "Weakenings" how many times the weakening procedure was invoked from the checking procedure. "T/O" indicates timeout. The timeout threshold was set to 3 hours (= 10800s). "CEX" indicates a bug found, "False" represents a false bug found, and "Pass" represents the case where a termination proof is found.

The difficulty in this code is that the spinlock `&devExt->SpinLock` is released and re-acquired within the loop that is draining the list of request packets. The motivation for releasing the lock is due to the performance cost of the procedure that "removes and completes" the I/O request packets (IRPs) in the queue: the programmer has decided that it is too expensive to call this operation while holding the lock. A fix is to create a new list, remove the elements from `ReadQueue`, place them into the new list, release the lock, and then "remove and complete" each request from the new queue. Examples 2 and 3 from Table 1 represent the two loops from this proposed fix.

8. Future work

We now refer to the current limitations of the proposed method that might be interesting to address in future work.

Supporting general liveness. In this paper we have restricted ourselves to termination only, and excluded arbitrary liveness properties [1] (*i.e.* fair termination) from consideration. However, it is likely that recent work on proving fair termination for sequential programs [20] can be adapted to the setting of thread-modular liveness checking.

```
while (!IsListEmpty(&devExt->ReadQueue)) {

  PLIST_ENTRY         ListElement;
  KIRQL               CancelIrql;

  ListElement=RemoveHeadList(
      &devExt->ReadQueue
      );

  Irp=CONTAINING_RECORD(ListElement,IRP,
            Tail.Overlay.ListEntry);

  IoAcquireCancelSpinLock(&CancelIrql);

  if (Irp->Cancel) {
    //  this one has been canceled
    Irp->IoStatus.Information=STATUS_CANCELLED;

    IoReleaseCancelSpinLock(CancelIrql);

    continue;
  }

  IoSetCancelRoutine( Irp, NULL);

  IoReleaseCancelSpinLock(CancelIrql);

  KeReleaseSpinLock( &devExt->SpinLock
                   , OldIrql);

  Irp->IoStatus.Information=0;

  RemoveReferenceAndCompleteRequest(
        devExt->DeviceObject, Irp,
       STATUS_CANCELLED);

  KeAcquireSpinLock( &devExt->SpinLock
                   , &OldIrql);
}
```

Figure 8. Fragment of Example 1 from Table 1 (a modem device driver) containing a concurrency/termination bug.

Unbounded threads, thread creation, thread destruction, etc. We have made the simplifying assumption that the set of threads in $\mathcal{P}$ is fixed. In principle our tool could be adapted to support an unbounded number of threads in the environment so long as they are drawn from a finite amount of code.

Synchronization primitives. For simplicity we have ignored several forms of synchronization (*e.g.* mutexes). In practice, however, these mechanisms are used in real programs and should be supported by our tool.

Stronger proof rule, stronger environment checking. Note that, when checking that T_i respects $\mathcal{A}$, we assume nothing of the environment that T_i executes in. Techniques exist to support this scenario (*e.g.* [33] and [35]), and could potentially be adapted to our setting.

Memory models. For simplicity we have largely ignored the complications due to non-atomic reads/writes to shared variables. As an example, our current setup would fail to prove the termination of T_1 from Figure 2 if the call to `InterlockedDecrement(&x)` were replaced with the assignment `x=x-1`, even though the program still does guarantee termination.

9. Conclusion

Concurrent programs are often designed such that the execution of certain subroutines (*i.e.* device driver dispatch routines) within threads will not diverge. We call this property thread-termination. We have described the first known program termination prover that natively supports proofs of thread-termination. When proving that the thread T_1 terminates, our prover attempts to find an abstract model of the environment that overapproximates the behavior of the other threads T_2, T_3, etc. This model allows us to perform the analysis thread-locally—we only consider each thread in isolation together with the environment abstraction.

The novelty of the work presented here is the method of strengthening the environment abstraction: it is incrementally computed using information mined from the examination of paths in the thread. The key idea is that we can take paths from the concurrent setting and prove them well-founded in the sequential setting. The witnesses to these proofs can lead to stronger environment abstractions.

We have demonstrated the practicality of the approach by implementing a tool and performing experiments with it on loops from device drivers. This evaluation represents the first known successful application of a termination prover for concurrent programs to industrial software. During the evaluation a number of driver loops were proved terminating, and several previously unknown bugs were also found.

Acknowledgments

We thank Josh Berdine, Georges Gonthier, Peter O'Hearn, and Viktor Vafeiadis for comments and suggestions. The third author is supported in part by Microsoft Research through the European Fellowship Programme.

References

[1] B. Alpern and F. Schneider. Defining liveness. *Information processing letters*, 21:181–185, 1985.

[2] I. Balaban, A. Cohen, and A. Pnueli. Ranking abstraction of recursive programs. In *VMCAI'06: Verification, Model Checking, and Abstract Interpretation*, 2006.

[3] T. Ball et al. Thorough static analysis of device drivers. In *EuroSys'06: European Systems Conference*, 2006.

[4] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *POPL'07: Principles of Programming Languages*, 2007.

[5] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV'06: International Conference on Computer Aided Verification*, 2006.

[6] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *FMICS'02: Formal Methods for Industrial Critical Systems*, 2002.

[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI'03: Programming Language Design and Implementation*, 2003.

[8] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV'06: International Conference on Computer Aided Verification*, 2006.

[9] A. Bradley, Z. Manna, and H. Sipma. Linear ranking with reachability. In *CAV'05: Computer-Aided Verification*, 2005.

[10] A. Bradley, Z. Manna, and H. Sipma. The polyranking principle. In *ICALP'05: International Colloquium on Automata, Languages and Programming*, 2005.

[11] A. Bradley, Z. Manna, and H. Sipma. Termination analysis of integer linear loops. In *CONCUR'05: Concurrency Theory*, 2005.

[12] A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI'05: Verification, Model Checking, and Abstract Interpretation*, 2005.

[13] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygia, and N. Sinha. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.

[14] E. Clarke, O.Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[15] E. M. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *VMCAI'06: Verification, Model Checking, and Abstract Interpretation*, 2006.

[16] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS'04: Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[17] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[18] M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV'02: Computer Aided Verification*, 2002.

[19] E. Contejean, C. Marché, B. Monate, and X. Urbain. Proving Termination of Rewriting with C*i*ME. In *WST'03: International Workshop on Termination*, 2003.

[20] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Vardi. Proving that software eventually does something good. In *POPL'07: Principles of Programming Languages*, 2007.

[21] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI'06: Programming Language Design and Implementation*, 2006.

[22] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI'05: Verification, Model Checking, and Abstract Interpretation*, 2005.

[23] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'78: Principles of Programming Languages*, 1978.

[24] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Journal on Theoretical Computer Science*, 338(1–3):153–183, 2005.

[25] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL'05: Principles of Programming Languages*, 2005.

[26] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN'03*, 2003.

[27] P. Godefroid. Partial-order methods for the verification of concurrent systems – an approach to the state-explosion problem. PhD thesis, 1994.

[28] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 1998.

[29] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *FSE'05*, 2005.

[30] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV'03*, 2003.

[31] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[32] B. Jacobs, K. R. M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM'05: Software Engineering and Formal Methods*, 2005.

[33] C. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

[34] V. Kahlon, A. Gupta, and N. Sinha. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In *CAV'06: International Conference on Computer Aided Verification*, 2006.

[35] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[36] Z. Manna and A. Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 1974.

[37] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[38] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'04: Verification, Model Checking, and Abstract Interpretation*, 2004.

[39] J. C. Reynolds. *The Craft of Programming*. London, 1981.

[40] A. Tiwari. Termination of linear programs. In *CAV'04: Computer Aided Verification*, 2004.