Abstract Threads

Shuvendu K. Lahiri[1], Alexander Malkis[2], and Shaz Qadeer[1]

[1] MSR Redmond
[2] University of Freiburg

Abstract. Verification of large multithreaded programs is challenging. Automatic approaches cannot overcome the state explosion in the number of threads; semi-automatic methods require expensive human time for finding global inductive invariants. Ideally, automatic methods should not deal with the composition of the original threads and a human should not supply a global invariant. We provide such an approach. In our approach, a human supplies a specification of each thread in the program. Here he has the freedom to ignore or to use the knowledge about the other threads. The checks on whether specifications of threads are sound as well as whether the composition of the specifications is error-free are handed over to the off-the-shelf verifiers. We show how to apply this divide-and-conquer approach to the interleaving semantics with shared variables communication where specifications are targeted to real-world programmers: a specification of a thread is simply another thread. The new approach extends thread-modular reasoning by relaxing the structure of the transition relation of a specification. We demonstrate the feasibility of our approach by verifying two protocols governing the teardown of important data structures in Windows device drivers.

1 Introduction

The motivation of our work is to verify the correctness of protocols embedded in large multithreaded programs. These programs typically access a variety of objects, including kernel resources and in-memory data structures; the protocols govern the policy for allocating, accessing, and freeing these objects. These protocols are hard to verify not only because of concurrency but also because the code implementing these protocols is typically spread over a large part of the program spanning multiple procedures and deep call chains. Invariably, there is no abstract formal model of these protocols; the code is the only artifact available for analysis.

A substantial amount of work has been done in formally verifying abstract protocol descriptions. However, all this work assumes that the protocol has somehow been extracted from the code implementing it. The extraction process is usually manual and hence error-prone. A bug in the manually-extracted model may not be a bug in the code; conversely, a proof of the manually-extracted model may not be a proof of the code.

Our paper contributes towards formalizing the model extraction problem, bringing much-needed rigor and automation to the process. We provide a simple compositional approach where the user provides an abstraction of each thread in the program. The abstraction of each thread is simply another thread, albeit one that may be significantly simpler due to the elimination of details irrelevant to the property of interest. Our notion of abstraction also allows an abstract thread to fail more often than its concrete counterpart, which is often useful for making the abstractions concise; in particular by avoiding the need to expose some local state of a thread that may indeed be relevant to the property of interest.

Given these abstract threads, the verification of an n-threaded concurrent program is decomposed into $n+1$ pieces – n local sequential checks that each thread conforms to its abstract thread, and the verification of the abstract multithreaded program obtained by composing the n abstract threads. We provide a method for checking that a concrete thread conforms to its abstract thread. The conformance check is reduced to checking the correctness of a sequential program, whose size is linear in the textual size of the concrete thread and quadratic in the number of local states of the abstract thread. In addition, the control structure of the sequential program is inherited from the thread; in particular it does not have any more loops than the underlying concrete thread. The sequential program can be automatically produced from the concrete and abstract threads.

Our method assists the model extraction process by telling whether the constructed models are sound and reporting an error when they are not; at the same time, the conformance checker uses powerful path-sensitive analysis and automated theorem provers, providing a precise way to check if a model abstracts the code, even in the presence of arithmetic and unbounded heap-allocated data structures. To the best of our knowledge, we present the first formal thread-by-thread modeling scheme.

The abstract threads serve as valuable contracts and documentation for the underlying code, and avoid performing a global analysis across the evolution of the underlying code of individual threads. The abstract multithreaded program can be considerably simpler compared to the original program and can be subjected to formal and rigorous analysis using existing techniques based on model checking [7], rely-guarantee reasoning [17] or thread-modular methods [12].

Proving correctness of a multithreaded program is much more efficient after model extraction. In general, model extraction can lead to exponential cost savings due to considering a simpler code. Even if a correctness proof unavoidably involves state explosion in the number of threads, model extraction can reduce the base by eliminating irrelevant local states, thus reducing the asymptotic verification time by an exponential factor.

In practice, the approach enables applying automatic verifiers and thus diminishes the total verification time. Since automatic verifiers cannot handle composition of large real-life threads, a user is doomed to fall back to manual global invariant specification; while creating small threads from large ones makes automatic tools usable. The price paid is identification of the relevant parts of a thread – and those parts may be taken without further inspection (of course, the user may wish to inspect them for further model reduction). This process requires less manual investment than manual specification of the whole global invariant, which would require both identification and inevitable thorough inspection of the relevant parts of a thread as a subtask.

To demonstrate the feasibility of our approach, we have applied it to the verification of two protocols governing the teardown of important data structures in Windows device drivers `battery` and `bluetooth`. The conformance check for each thread is implemented using the Boogie [2] verifier and the abstract multithreaded programs are checked using Boogie and SPIN [15].

2 Programs, executions and specifications

A *thread* T is a tuple $(Global, Local, \to, Init, Wrong)$ where
- $Global$ is a set of global states;
- $Local$ is a set of local states;
- $\to \subseteq (Global \times Local)^2$ is the transition relation;
- $Init \subseteq Local$ is the set of initial local states;
- $Wrong \subseteq Global \times Local$ is the set of error states;

This “local” error state definition is targeted towards the naturally given specifications: `assert` statements in the program code and implicit language constraints like absence of NULL-pointer dereferences. Checking general safety properties is reducible to local error checks.

We use letters g and h to denote global states and symbols $\overline{g}$ and $\overline{h}$ to denote sequences of global states. Similarly, we use letters l and m to denote local states and symbols $\overline{l}$ and $\overline{m}$ to denote sequences of local states.

A phased execution of a thread comprises an alternating sequence of transitions of this thread and transitions of the environment of this thread. The number of thread transitions, which is equal to the number of environment transitions, is the length of the execution. A phased execution of length three is shown in Figure 1; thread transitions are depicted by solid arrows going horizontally and environment transitions are depicted by dashed arrows going vertically.

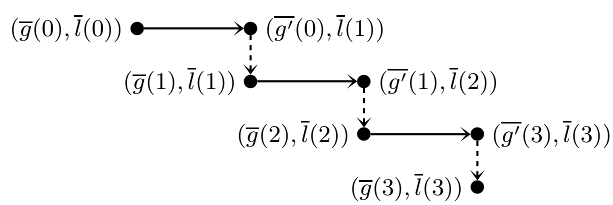


Fig. 1. A phased execution of length three.

Formally, a *phased execution* of T of length p is a triple $(\overline{g}, \overline{g'}, \overline{l})$, where $\overline{g}$ and $\overline{l}$ are sequences of length $p+1$ and $\overline{g'}$ is a sequence of length p, such that $\overline{l}(0) \in Init$, $(\overline{g}(j), \overline{l}(j)) \notin Wrong$, and $(\overline{g}(j), \overline{l}(j)) \to (\overline{g'}(j), \overline{l}(j+1))$ for all j such that $0 \le j < p$.

Let $T = (\mathit{Global}, \mathit{Local}, \to, \mathit{Init}, \mathit{Wrong})$ and $T^{\#} = (\mathit{Global}, \mathit{Local}^{\#}, \to^{\#}, \mathit{Init}^{\#}, \mathit{Wrong}^{\#})$ be some threads over the same set of shared states. Let $e = (\overline{g}, \overline{g'}, \overline{l})$ be a phased execution of T of length p and $e^{\#} = (\overline{h}, \overline{h'}, \overline{m})$ a phased execution of $T^{\#}$ of length q. Then $e^{\#}$ *abstracts* e if $q \leq p$ and all of the following conditions hold:

- $\overline{g}|_{q+1} = \overline{h}$ and $\overline{g'}|_q = \overline{h'}$;
- if $(\overline{g}(p), \overline{l}(p)) \in \mathit{Wrong}$ or $q < p$, then $(\overline{h}(q), \overline{m}(q)) \in \mathit{Wrong}^{\#}$.

(Here, $\overline{x}|_y$ is the prefix of $\overline{x}$ of length y. For presentation purposes the conditions are kept simple; however, it is possible to relax them by allowing e or $e^{\#}$ to do some internal actions.) Intuitively, the execution $e^{\#}$ must be a prefix of e and must end in an error state if either e ends in an error state or $e^{\#}$ is shorter than e. An interesting aspect of our definition is that $e^{\#}$ is allowed to go wrong earlier than e. We show below using an example how this feature of our definition allows concise abstractions. Thread $T^{\#}$ is an *abstraction* of thread T if every phased execution of T is abstracted by some phased execution of $T^{\#}$.

A *multithreaded program* P is a tuple $(T_i)_{i \in \mathit{Tid}}$, where Tid is a set of thread identifiers and $T_i = (\mathit{Global}, \mathit{Local}, \to_i, \mathit{Init}_i, \mathit{Wrong}_i)$ is a thread. Let $\mathit{Locals} = \mathit{Tid} \to \mathit{Local}$ be the set of all tuples of local states of threads in Tid. A state s of P is a tuple $(g, \mathit{ls}) \in (\mathit{Global} \times \mathit{Locals})$. The state (g, ls) is an *initial* state if $\mathit{ls}[i] \in \mathit{Init}_i$ for all $i \in \mathit{Tid}$. The *transition relation* of P is $\longrightarrow \subseteq (\mathit{Global} \times \mathit{Locals})^2$, defined by

$$
\begin{aligned}
&(g, \mathit{ls}) \longrightarrow (g', \mathit{ls}') \quad :\Leftrightarrow \\
&\exists\, i \in \mathit{Tid} : (g, \mathit{ls}[i]) \to_i (g', \mathit{ls}'[i]) \wedge \forall\, j \in \mathit{Tid} \setminus \{i\} : \mathit{ls}[j] = \mathit{ls}'[j]\,.
\end{aligned}
$$

An *execution* of P is a sequence $\overline{s}$ of length $k > 0$ such that $\overline{s}(0)$ is an initial state and $\overline{s}(j) \longrightarrow \overline{s}(j+1)$ for all j such that $0 \leq j$ and $j+1 < k$. The program P *goes wrong* from a global state g if there exist $\mathit{ls}, \mathit{ls}' \in \mathit{Locals}$ and $g' \in \mathit{Global}$ such that all of the following conditions hold:

- (g, ls) is an initial state of P;
- there is an execution of P from (g, ls) to (g', ls');
- $(g', \mathit{ls}'[i]) \in \mathit{Wrong}_i$ for some $i \in \mathit{Tid}$.

Our definition of abstraction is sound for modular reasoning. If each thread in a multithreaded program P is abstracted by a corresponding thread in a program $P^{\#}$, then it suffices to prove $P^{\#}$ correct in order to prove P correct. This claim is captured by the following theorem.

Theorem 1 (Soundness). *Let $P = (T_i)_{i \in \mathit{Tid}}$ and $P^{\#} = (T_i^{\#})_{i \in \mathit{Tid}}$ be multithreaded programs over the set Global of shared states such that $T_i^{\#}$ is an abstraction of T_i for all $i \in \mathit{Tid}$. Then, for all $g \in \mathit{Global}$, if P goes wrong from g, then $P^{\#}$ also goes wrong from g.*

2.1 Example

Consider the multithreaded program P in Figure 2. This program has two threads and a single shared variable `g`. We assume that every line in the program is executed atomically. Suppose we wish to prove that the assertion in the program does not fail whenever we execute P from a global state satisfying `g>0`.

```
Thread A                      Thread B

int x;                        int y;
x := 1;                       y := 1;
while (*) {                   while (*) {
   if (*) {                      y := y+1;
     x := g;                  }
   } else {                   g := y;
     x := x+1;                assert g>0;
   }
}
g := x;
```

Fig. 2. Multithreaded program P

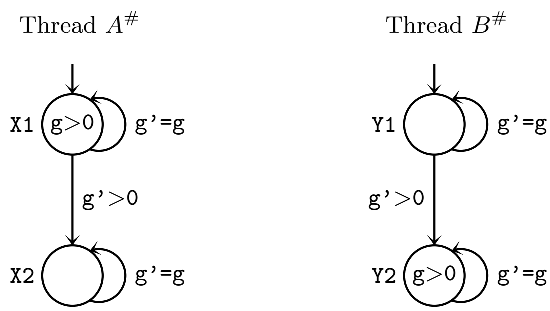


Fig. 3. Multithreaded program $P^{\#}$

Instead of proving this property directly on P, we would instead like to prove it on the simpler program $P^{\#}$ in Figure 3. Since $P^{\#}$ is intended to be a specification, it is written in the style of a state machine. The labels `X1, X2, Y1, Y2` are local states of $A^{\#}$ and $B^{\#}$, the predicate drawn in a local state is an assertion (defaults to `true` if none given) and the predicate on an edge indicates the transition relation for that edge. For example, an execution beginning at `X1` goes wrong if `g>0` is false; otherwise, either the program location and the value of `g` remain unchanged or the program location changes to `X2` and the value of `g` is updated to some number greater than zero. The initial local states of the threads are `X1` for $A^{\#}$ and `Y1` for $B^{\#}$. Note that the local state of threads in $P^{\#}$ are finite, while each thread in P has a local integer variable in addition to the set of program locations.

Each phased execution of thread A is abstracted by a phased execution of thread $A^{\#}$. Every transition of A before the update `g := x` is "simulated" by the transition of $A^{\#}$ that goes from `X1` to `X1`. The update `g := x` is "simulated" by the transition $g' > 0$ from `X1` to `X2`. The correspondence between B and $B^{\#}$ is similar. The assertion at the end of B carries over to the assertion in state `Y2`. The next section will present our conformance checking algorithm for formally proving that A is abstracted by $A^{\#}$ and B is abstracted by $B^{\#}$.

Note that while there is no assertion in A, we have introduced an assertion in $A^{\#}$ in the state `X1`. This assertion is essential for conformance checking to work, since otherwise thread A would have no assumption about the values read from g and could assign also negative values to g, which is in turn not modeled by $A^{\#}$.

Note that even though A is a small program, $A^{\#}$ is considerably simpler than A. Although it has more assertions, its local state has become finite by the elimination of the variable `x`. In fact, the introduced assertion is the key reason for being able to eliminate the variable `x`.

For demonstration purposes, consider a variant $\tilde{A}$ of A in which `x:=1` is replaced by `x:=-1`. Then $\tilde{A}$ could update g to a negative value, and since $A^{\#}$ can update g only to a positive value, $A^{\#}$ would not be an abstraction of $\tilde{A}$.

3 Conformance checker

Now we show how to check that a thread is abstracted by another thread.

Let $\mathcal{P}(X)$ denote the powerset of a set X.

A *sequential program* is a tuple $(Z, \rightsquigarrow, \mathit{Start}, \mathit{Error})$ where

- Z is a set of states;
- $\rightsquigarrow \subseteq Z^2$ is the transition relation;
- $\mathit{Start} \subseteq Z$ is the set of initial states;
- $\mathit{Error} \subseteq Z$ is the set of error states.

An *execution* of a sequential program is a nonempty sequence $\overline{z}$ of states of finite length k such that $\overline{z}(0) \in \mathit{Start}$ and $\overline{z}(j) \rightsquigarrow \overline{z}(j+1)$ for all j such that $0 \leq j$ and $j+1 < k$. An execution is called *failing* if any of its states is in Error. A sequential program is *correct* if it has no failing execution.

Let $T = (\mathit{Global}, \mathit{Local}, \rightarrow, \mathit{Init}, \mathit{Wrong})$ and $T^{\#} = (\mathit{Global}, \mathit{Local}^{\#}, \rightarrow^{\#}, \mathit{Init}^{\#}, \mathit{Wrong}^{\#})$ be two threads. Our solution to the problem of checking that T is abstracted by $T^{\#}$ is encoded as a sequential program $C(T, T^{\#})$. This program simultaneously runs both T and $T^{\#}$ checking that each step of T is "simulated" by the corresponding step of $T^{\#}$. Since $T^{\#}$ is potentially nondeterministic, a partial execution of T can be "simulated" by multiple executions of $T^{\#}$. Consequently, a state of $C(T, T^{\#})$ is a pair (l, F) from the set $\mathit{Local} \times \mathcal{P}(\mathit{Local}^{\#})$. The first component l is the state of T. The second component F is the set of states of $T^{\#}$ that are candidates for "simulating" future behaviors of T from l. Our construction provides the guarantee that $C(T, T^{\#})$ goes wrong iff T is not abstracted by $T^{\#}$.

We now provide a formal definition of $C(T, T^{\#})$. For each $l \in \mathit{Local}$, let $W(l) = \{g \in \mathit{Global} \mid (g, l) \in \mathit{Wrong}\}$. Similarly, for each $m \in \mathit{Local}^{\#}$, let $W^{\#}(m) = \{g \in \mathit{Global} \mid (g, m) \in \mathit{Wrong}^{\#}\}$. For each $F \in \mathcal{P}(\mathit{Local}^{\#})$, let $W^{\#}(F) = \bigcup_{m \in F} W^{\#}(m)$. A *conformance checker* $C(T, T^{\#})$ is a sequential program $(Z, \rightsquigarrow, \mathit{Start}, \mathit{Error})$ where

- $Z = \mathit{Local} \times \mathcal{P}(\mathit{Local}^{\#})$;
- $\mathit{Start} = \mathit{Init} \times \{\mathit{Init}^{\#}\}$;
- $\mathit{Error} = (\mathit{Local} \times \{\emptyset\}) \cup \{(l, F) \in \mathit{Local} \times \mathcal{P}(\mathit{Local}^{\#}) \mid W(l) \not\subseteq W^{\#}(F)\}$;

– $\rightsquigarrow$ is defined by

$$\frac{\begin{array}{c} l, l' \in \mathit{Local} \quad F, F' \subseteq \mathit{Local}^{\#} \\ \exists\, g, g' \in \mathit{Global} : g \notin W^{\#}(F) \text{ and } (g, l) \rightarrow (g', l') \text{ and} \\ F' = \{m' \mid \exists\, m \in F : (g, m) \rightarrow^{\#} (g', m')\} \end{array}}{(l, F) \rightsquigarrow (l', F')}$$

The definition of *Error* and $\rightsquigarrow$ are the most interesting and subtle parts of the definition of $C(T, T^{\#})$. The set *Error* is the union of two parts, each corresponding to a different reason why T might not be abstracted by $T^{\#}$. Consider an element $(l, \emptyset)$ of the first part. The conformance checker makes a transition to this state if it arrives in a state (x, F) and there is some transition of T out of the local state x that cannot be "simulated" by any transition of $T^{\#}$ out of any local state in F. Now, consider an element (l, F) of the second part satisfying $W(l) \not\subseteq W^{\#}(F)$. If the conformance checker arrives in this state, then T can go wrong from l but $T^{\#}$ cannot go wrong from any candidate state in F, again violating a requirement for abstraction.

Having understood the definition of *Error*, it is simple to understand the definition of $\rightsquigarrow$. We create a transition $(l, F) \rightsquigarrow (l', F')$ only when there exist g, g' such that T can make a transition from (g, l) to (g', l'). Here, we only need to pick those states g from which it is not possible for $T^{\#}$ to go wrong from a local state in F. The reason is that if $T^{\#}$ can go wrong from g, the "simulation" process can stop because we have discovered an erroneous execution in the abstraction. We collect in F' all those local states transitions to which can "simulate" the transition (g, l) to (g', l') of T.

There are two important observations about our conformance checker. First, its control structure is inherited from the thread T. Any loops in T get carried over to $C(T, T^{\#})$; moreover, if T is loop-free then so is $C(T, T^{\#})$. Second, the state of $C(T, T^{\#})$ is independent of the global state set *Global*. Essentially, the global state gets existentially quantified at each step of the conformance checking computation. This property allows us to write loop invariants for $C(T, T^{\#})$ without worrying about the behavior with respect to the global state.

There are a few special cases for which the conformance checker becomes simpler. First, if the set $\mathit{Local}^{\#}$ of abstract local states is finite, then the (in general, unbounded) quantification implicit in the definition of *Error* and the calculation of F' in the definition of $\rightsquigarrow$ become finite. The conformance checker can simply enumerate the set of abstract local states allowing the assertion logic of the sequential program to become simpler. In addition, if the concrete thread T is either finite-state or loop-free, then the correctness of the conformance checker can be verified fully automatically, using a finite-state model checker or an automated theorem prover, respectively.

The correctness of our conformance checker is captured by the following theorem.

Theorem 2. *Let* $T = (Global, Local, \rightarrow, Init, Wrong)$ *and* $T^{\#} = (Global, Local^{\#}, \rightarrow^{\#}, Init^{\#}, Wrong^{\#})$ *be threads over a nonempty set of shared states Global. Then* $T^{\#}$ *is an abstraction of* T *if and only if the conformance checker* $C(T, T^{\#})$ *is correct.*

3.1 Instrumentation

Now we show how to check abstraction when threads are given in a textual form rather than as transition systems.

The conformance checker is implemented by instrumenting the source code of the concrete thread. We now describe this instrumentation for the case when the local state of the abstract thread is finite. We make the following simplifying assumptions about the code of the concrete thread. First, we assume that every statement is executed atomically: all statements have been split into atomic substatements in a standard way which depends on the platform on which the code is executed. Second, we assume that all conditionals have been eliminated using assume statements and standard control-flow-graph transformations. Thus, apart from the usual control flow, we only have assume, assert, and assignment statements in the code. Third, we assume that the program has a single global variable named v; the extension for multiple variables is straightforward.

Finally, we assume that the abstract thread is provided as a state machine (as in Figure 3) comprising a finite set of nodes N with edges among them. Each node x is labeled with a predicate $A(x)$ over the variable v capturing the assertion in that state. An edge from node x to node y is labeled with a predicate $T(x, y)$ over variables v and v' capturing the transition from local state x to local state y. The set $I \subseteq N$ is the set of initial states.

We are now ready to describe the instrumentation. We introduce a bookkeeping variable u for keeping a temporary copy of v. We also introduce a map variable $F : N \rightarrow Boolean$ to model the set of locations of the abstract thread. We insert the following initialization code at the beginning of the concrete thread.

havoc v; // assign any value to v nondeterministically
$u := v$;
$F := \lambda y \in N.\ y \in I$;

Next, we replace each non-assert statement *st* in the program with the following code fragment.

(1) *assert* $\bigvee_{x \in N} F[x]$;
(2) *assume* $\bigwedge_{x \in N} F[x] \Rightarrow A(x)$;
(3) *st*;
(4) $F := \lambda y \in N.\ \bigvee_{x \in N} F[x] \wedge T(x, y)[u/v, v/v']$;
(5) *havoc* v;
(6) $u := v$;

This instrumentation preserves the invariant that upon entry to each instrumentation code block, variables u and v are identical and unconstrained. Clearly, this property is true initially; lines 5 and 6 ensure this property upon exit from the code block. Line 1 asserts that the set F is nonempty. Line 2 assumes the assertions at each location in F. If any of these facts do not hold then the abstract execution can go wrong and the "simulation" check has succeeded. Line 3 simply executes the statement *st* and line 4 computes the new value of the set F by enumerating over all pairs of abstract nodes.

Finally, each original-code assertion *assert*(ϕ) (which specifies a property to be proven) is replaced by the check

$$assert\ \neg\phi \Rightarrow \bigvee_{x \in N} F[x] \wedge \neg A(x);$$

The check asserts that when a concrete execution goes wrong, at least one abstract execution should also go wrong.

From this instrumentation technique, it is clear that the conformance checker inherits the control structure of the concrete thread. Furthermore, the textual size of the checker is linear in the size of the concrete thread and quadratic in the size of the abstract thread. The instrumentation is a simple syntactical operation and can be performed fully automatically. If the local state of the abstract thread is finite, the analysis of the conformance checker can be automatized to the same level as that of the concrete thread, viewed as a sequential program.

4 Experiments

We demonstrate our approach on two drivers from the Windows operating system.

4.1 Bluetooth driver

The concrete multithreaded program in Figure 4 consists of a thread `PnpStop` which unloads the driver and n threads `PnpAdd` which process data. The shared variables together with their initial values are `pendingIO = 1`, `stoppingFlag = stoppingEvent = stopped = 0`.

The property to be proven is the correct teardown, i.e. that in no execution a "worker" thread should try to access data after `stopped` flag has been raised by the "unload" thread.

SPIN cannot check the composition of 9 threads executing Fig. 4, as we will see. To increase the number of verifiable threads, it is reasonable to simplify the Bluetooth code. We simplified the code by eliminating the local variable `status` and merging some local states to reduce their number. A possible resulting abstract program is given in Figure 5.

We use the following shorthands: sF = `stoppingFlag`, pIO = `pendingIO`, sE = `stoppingEvent`, st = `stopped`. Further, there is a hidden idle transition associated with each node, labeled with $pIO = pIO' \wedge sF = sF' \wedge sE = sE' \wedge st = st'$. The notation $\langle\phi\rangle_{X,Y}$ is a shorthand for $\phi \wedge \bigwedge_{v \in \mathrm{Var} \setminus \{X,Y\}} v = v'$, i.e., that all variables except X and Y remain unchanged. The formulas in the nodes denote assertions. If a formula is missing, it defaults to *true*.

```
// ‘‘Unload’’ thread
void PnpStop() {
     stoppingFlag=1;
     IoDecrement();
     assume(stoppingEvent);
     // release allocated resources;
     stopped=1;
}
```

```
// ‘‘Worker’’ thread
void PnpAdd() {
     int status;
     status = IoIncrement();
     if(status>0) {
         // do work here
         assert(!stopped);
     }
     IoDecrement();
}
```

```
int IoIncrement() {
     int status;
     pendingIO++;
     if(stoppingFlag)
         status=-1;
     else
         status=1;
     return status;
}
```

```
void IoDecrement() {
     pendingIO--;
     if(!pendingIO)
         stoppingEvent=1;
}
```

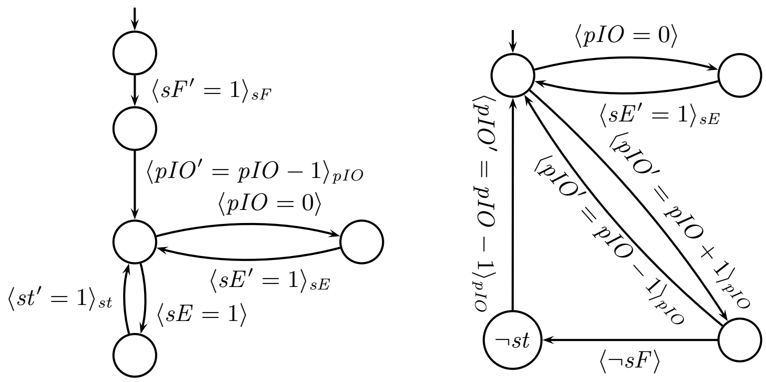
Fig. 4. Bluetooth driver model consisting of a single “unload” thread `PnpStop` and n “worker” threads `PnpAdd`.

Such simplification looks easy but is error-prone: without a correctness proof one never knows whether the party (a human or a tool) that did the simplification has really produced sound abstractions. Our approach automatically creates a formal proof of abstraction soundness.

To encode the driver model and its abstraction, we used the Boogie modeling language equipped with the Boogie verifier and Z3 theorem prover [10]. The conformance check succeeded fully automatically. Boogie also allowed automatic bounded verification of the composition of thread abstractions for one “unload” and two “worker” threads. All the mentioned checks together succeeded within 33 seconds. The concrete multithreaded program in Figure 4 and, separately, the abstract multithreaded program in Figure 5 were fed to the SPIN tool. Proofs by exhaustive search needed following times in seconds:

Threads	1	2	3	4	5	6	7	8	9	10	11
Concrete program	0	0	0	0.0225	0.1833	1.56	19.26	158.66	n/a	n/a	n/a
Abstract program	0	0	0	0	0.018	0.12	0.314	1.246	5.812	39.04	190.6

Zero means that the time was so small that it was below the measurement precision, “n/a” means that SPIN exceeds the 2GB memory limit. The asymptotic verification times for the concrete program and abstract programs are exponential, as expected for the general-purpose uninformed tool, with experimentally determined bases ≈ 9.3 and ≈ 4.9. Abstraction allowed verification of more threads in less time. The exponential speedup is $\approx 1.9^n$.



(a) Abstract "unload" thread (b) Abstract "worker" thread

Fig. 5. Abstract bluetooth driver model.

4.2 Battery driver

The property to verify is correct teardown, i.e. once the dispatch routines have started the "worker" threads, and the "unload" thread has freed the data structures of the driver, no "worker" thread should try to access these data structures any more. We examine a simple version of the driver by inlining procedures and modeling the fields of the heap allocated data structure as scalar variables.

Figure 6 shows the simplified code of the "unload" and "worker" threads. A star ($*$) represents a nondeterministic value, and the variable `stopped` indicates that the object has been freed.

To ensure correct teardown, we put an implicit assertion $\neg$`stopped` before each access to a shared variable (except `NumQueuedWorkers`).

Initially, `WorkerActive` $= 0$, `WantToRemove` $=$ FALSE, `ReadyToRemove` $=$ FALSE, `InUseCount` $= 2$, `NumQueuedWorkers` $= 0$. When a worker thread is scheduled for execution by a dispatch routine (which is not depicted here), the routine increments the `WorkerActive` counter. When a copy of the worker thread is about to quit, the worker thread decrements `WorkerActive`. The variable `InUseCount` models a reference count of the number of threads accessing a shared object and is incremented by a thread before accessing the object and decremented later. Furthermore, if the removal is signaled by `ReadyToRemove`, threads decrement `InUseCount` and try to quit themselves. We made one simplifying assumption that the decrement of `InUseCount` to 0 and the signaling of `ReadyToRemove` happens atomically. This simplification is justified because the action that sets `ReadyToRemove` to true commutes to the left of all concurrent actions of other threads.

The abstract worker thread is shown in Figure 7. We introduce shorthands: R(eadyToRemove), I(nUseCount), N(umQueuedWorkers), W(orkerActive) and S(topped). In the following pictorial representation, each assertion is implicitly conjoined with the common part: $I \geq 0 \wedge W \geq 0 \wedge N \geq 0 \wedge (R \Rightarrow I = 0) \wedge$

"Unload" thread	"Worker" thread

<pre>WantToRemove=TRUE;
if(1 == ++WorkerActive) {
  if(*) {
    if(0 == --InUseCount)
      ReadyToRemove=TRUE;
  } else {
    ++NumQueuedWorkers;
    // Work to do,
    // start working thread.
  }
}
if(0 < --InUseCount)
  await(&ReadyToRemove);
stopped=TRUE;</pre>
	<pre>atomic {
	  await(NumQueuedWorkers>0);
	  NumQueuedWorkers--;
	}
	unsigned long i;
	while(TRUE) {
	  if(WantToRemove) {
	    if(0 == --InUseCount)
	      ReadyToRemove=TRUE;
	    break;
	  }
	  if(*) {
	    ++InUseCount;
	    if(WantToRemove)
	      --InUseCount;
	    else
	      --InUseCount;
	  }
	  i = --WorkerActive;
	  if(0==i) break;
	  if(1!=i) WorkerActive=1;
	}</pre>

Fig. 6. Battery driver: "unload" and "worker" threads.

($N > 0 \Rightarrow \neg S \wedge W > 0$). Further, there is a hidden idle transition associated with each node, labeled with $R = R' \wedge I = I' \wedge N = N' \wedge W = W' \wedge S = S'$. One interesting thing to observe is that the local variable `i` is not present in the abstract specification for the worker thread. This is important to make the set of local states of the worker thread specification finite, thus enabling us to leverage the instrumentation provided in Section 3.1.

We encoded the concrete and the abstract threads into the Boogie modeling language. We supplied loop invariants for the loops in the conformance checker manually. The corresponding conformance checkers were proven correct using the Z3 theorem prover in around two minutes for all the threads.

The abstract and concrete programs were also written in the Promela modeling language after manual abstraction of the unbounded `WorkerActive` variable (one can replace it by the predicate `WorkerActive` > 0). The resulting code was fed into SPIN, which created proofs by exhaustive search in the following time in seconds:

Threads	1	2	3	4	5	6	7	8	9	10
Concrete program	0	0	0.08	1.042	7.564	54.28	242.4	929	t/o	t/o
Abstract program	0	0	0	0	0.11	0.51	1.3	3.29	7.49	19.2

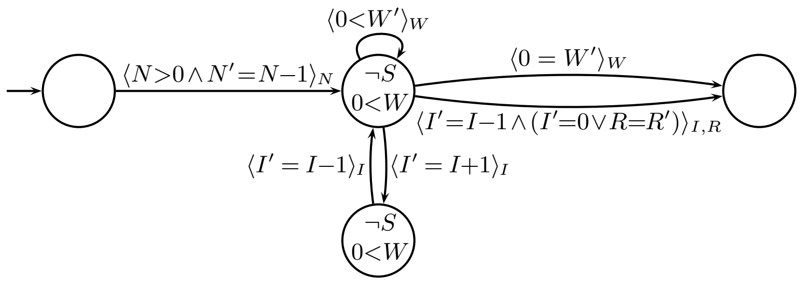


Fig. 7. Specification of worker thread

The symbol "t/o" means SPIN has exceeded the time bound of 20 minutes. The empirical asymptotic runtimes are $\approx 3.832^n$ and $\approx 2.227^n$. Abstraction thus allowed an exponential speedup of $\approx 1.72^n$.

5 Related Work

In this work, we presented a compositional approach for analyzing real-world multithreaded programs based on abstract threads. Our main contribution is in providing a framework that allows the user to construct and check abstractions of each thread and verify the composition of the abstract multithreaded program. The approach can be seen as a semantic method for simplifying each thread before performing an analysis of the multithreaded program. We also believe that the abstract threads are intuitive specifications of each thread for a developer because they allow the user to express complex control flow required to capture many real-life protocols.

We can view existing work on verifying multithreaded programs as complementary to our work — we can use any one of them for verifying our abstract multithreaded program and these techniques can use our formalism to simplify the input programs. Existing approaches to verifying multithreaded programs use methods based on inductive invariants [21,9], induction [20], rely-guarantee reasoning [17], partial invariants [24], thread-modular reasoning [12,19,8], model-checking [7], concurrent data-flow analysis [26,13,6] or even bounded analysis [22]. The analysis methods differ in the level of automation and completeness of checking the underlying system. Model-checking based methods are automatic for finite state models extracted manually or as a result of abstraction [14,4], but suffer from state explosion or imprecision in the presence of complex data types. Concurrent data-flow analysis engines extend sequential data-flow analysis in the presence of concurrency, but are restricted to particular analysis domains.

Our method is closest to the class of works based on rely-guarantee mechanism; these approaches allow the user to specify rely-guarantee contracts for each thread; however the annotation can be complex for real-life programs.

We are the first, to the best of our knowledge, to introduce our abstraction relation between threads. Classical simulation of [1] doesn't separate shared and local states, [23] uses bisimulation. Closer simulation relations can be found in process algebras [16,5,18], where a type, which is written in π-calculus or CCS, represents an abstraction of a process, which is written in π-calculus.

Two-level verification occurs in [4], where message-passing communicating processes get abstracted to pushdown automata via boolean abstraction. Apart from the difference in the communication model, we allow richer abstraction, since it is possible to encode boolean programs as abstract threads, but not every thread can be encoded as a boolean program due to the possible presence of unbounded data.

The use of ownership methodology in Spec# [3] and separation logic [25] have the potential to make the specifications more manageable by restricting annotations to the local heap. Flanagan et al. [11] allow linearly ordered control states in the specification of a thread, but do not allow rich control structure of abstract threads. Their "method may be extended to more general abstractions ... at the cost of additional complexity" (p. 166). Our method is such an extension in the call-free case.

6 Conclusion

In this work, we presented a compositional framework to check the correctness of a multithreaded program. We believe that the notion of abstract threads provides an intuitive as well as an expressive formalism for describing models of real-life multithreaded programs. We have illustrated the feasibility of our approach by studying two protocols present in real-life device drivers.

There are several directions in which we are extending the current work. Currently, procedures are treated only as control flow structures; we believe that our method can deal with procedure specifications naturally. Second, we are working on overcoming the restrictions of the real-world model-checkers, e.g. assist the tool in handling loops (as in Boogie) or unbounded variables (as in SPIN). Third, we are exploring techniques that assist a human in creating abstract threads. Finally, we are targeting more real-world examples to evaluate our method.

References

1. C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.
2. M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Objects and Components (FMCO '05)*, pages 364–387, 2005.
3. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, LNCS 3362, pages 49–69, 2005.
4. S. Chaki, E. M. Clarke, N. Kidd, T. W. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 3920, pages 334–349, 2006.

5. S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *POPL*, pages 45–57, 2002.
6. R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Programming Language Design and Implementation (PLDI)*, pages 316–326. ACM, 2008.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 2000.
8. A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. In *CAV*, LNCS 4590, pages 55–67. Springer, 2007.
9. P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Program Construction Techniques*, pages 243–271. Macmillan, 1984.
10. L. de Moura and N. Bjørner. Efficient incremental E-matching for SMT solvers. In *Conference on Automated Deduction (CADE '07)*, LNCS 4603, 2007.
11. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.
12. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, LNCS 2648, pages 213–224. Springer, 2003.
13. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *Programming Language Design and Implementation (PLDI)*, pages 266–277. ACM, 2007.
14. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In W. Pugh and C. Chambers, editors, *PLDI*, pages 1–13. ACM, 2004.
15. G. Holzmann. The SPIN model checker: Primer and reference manual. Addison-Wesley, ISBN 0-321-22862-6, `http://www.spinroot.com`.
16. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *POPL*, pages 128–141, 2001.
17. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
18. N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the pi-calculus. *Logical Methods in Computer Science*, 2(3), 2006.
19. A. Malkis, A. Podelski, and A. Rybalchenko. Precise thread-modular verification. In *SAS*, volume 4634 of *LNCS*, pages 218–232. Springer, 2007.
20. K. L. McMillan, S. Qadeer, and J. B. Saxe. Induction in compositional model checking. In *CAV*, LNCS 1855, pages 312–327. Springer, 2000.
21. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.
22. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *CAV*, volume 3576 of *LNCS*, pages 82–97. Springer, 2005.
23. C. Röckl and J. Esparza. On the mechanized verification of infinite systems, 2000. SFB 342 Final Colloquium.
24. H. Seidl, V. Vene, and M. Müller-Olm. Global invariants for analyzing multi-threaded applications. In *In Proc. of Estonian Academy of Sciences: Phys., Math*, pages 413–436, 2003.
25. V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *Concurrency Theory (CONCUR)*, LNCS 4703, pages 256–271, 2007.
26. E. Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *POPL*, pages 27–40, 2001.

A Embedding Thread-Modular Reasoning

Now we show that any call-free finite-state program with thread-modular proof in the sense of [11] also admits abstract threads which suffice to prove the property such that the size of the abstract threads is linear in the size of the thread-modular specification and quadratic in the number of shared states.

Simplifying [11], assume a single procedure. Its control location set is *Local* and the transition relation of the procedure for each $t \in \mathit{Tid}$ is $\to_t \subseteq (\mathit{Global} \times \mathit{Local})^2$. Let $0 \in \mathit{Local}$ be the initial control location, $\mathit{pre}, \mathit{post} \subseteq \mathit{Global}$ the pre- and postconditions of the procedure. One has to prove that the concurrent execution of all $\to_t$ ($t \in \mathit{Tid}$) starting from *pre* respects an invariant $I \subseteq \mathit{Global}$ and that each thread satisfies *post* when it terminates. A thread-modular proof of this fact requires the human to write a specification $(A, E) \in ((\mathcal{P}(\mathit{Global}^2))^*)^{Tid} \times (\mathcal{P}(\mathit{Global}^2))^{Tid}$, where E denotes the environment assumptions as transition predicates and A denotes the abstractions of the threads. The abstraction is a statement sequence interleaved with the stuttering steps: $A_t = \mathit{pre}?\langle\mathtt{true}\rangle; (I?K^*; I?Y_1); \ldots; (I?K^*; I?Y_m); I?K^*; \mathtt{true}?\langle \mathit{post}\rangle$ where K is the stuttering relation and $p?X$ means a state assertion p together with a transition relation X ($t \in \mathit{Tid}$). If for each $t \in \mathit{Tid}$, the program in which the tth thread is $\to_t$ and all the other threads are E_t^* is simulated by the program in which the tth thread is A_t and all the other threads are E_t^*, then the original program is correct.

Now we transform the threads and specifications to our setting. The concrete threads are modeled straightforwardly as $T_t = (\mathit{Global}, \mathit{Local}, \to_t, \{0\}, \mathit{Wrong})$ where $\mathit{Wrong} = \{(g, l) \in \mathit{Global} \times \mathit{Local} \mid g \notin I \text{ or } (l = 0 \text{ and } g \notin \mathit{pre})\}$ ($t \in \mathit{Tid}$). The abstract threads are $T_t^\# = (\mathit{Global}, \mathit{Local}^\#, \to_t^\#, \mathit{Init}^\#, \mathit{Wrong}_t^\#)$ ($t \in \mathit{Tid}$) where

- $\mathit{Local}^\# = \{0, 1, 2, \ldots, m+1\} \times \mathit{Global}$
- $(g, (l, h)) \to_t^\# (g', (l', h'))$ iff $g' = h'$ and $((l = l'$ and $g = g')$ or $(l' = l+1 \leq m$ and $(g, g') \in Y_{l+1})$ or $(l = m$ and $l' = m+1$ and $g' \in \mathit{post}))$
- $\mathit{Init}^\# = \{0\} \times \mathit{pre}$
- $\mathit{Wrong}_t^\# = \{(g, (l, h)) \in \mathit{Global} \times (\mathit{Local} \times \mathit{Global}) \mid g \notin I$ or $(l = 0$ and $g \notin \mathit{pre})$ or $(h, g) \notin E_t^*\}$.

For each $t \in \mathit{Tid}$, the abstract thread $T_t^\#$ should mimic the transitions and specifications given by A_t and E_t. For that, $T_t^\#$ must fail at least when A_t fails. Moreover, the abstract thread has to fail when the environment violates its specification. To track the change of the shared state by the environment, the abstract thread keeps a copy of the shared state in its own local state. A state of the abstract thread is thus $(g, (l, h))$ where g is a shared state, l the local abstract program counter and h locally stores some shared state.

The environment of the abstract thread may change g, but keeps the copy h in the local part unchanged. The abstract thread compares the shared state h before the environment transitions with the current state g and fails if the environment doesn't behave according to E_t. This is taken care of by the definition of $\mathit{Wrong}_t^\#$.

The transition relation $\to_t^\#$ mimics all the transition of $A_t^\#$ and additionally saves the current shared state at each step.

B Proofs

A *schedule* of an execution $(\overline{g}(j), \overline{ls}(j))_{j<p}$ of a multithreaded program $P = (T_t)_{t \in Tid}$ where $T_t = (Global, Local, \to_t, Init_t, Wrong_t)$ for all $t \in Tid$ is a map $s\colon p-1 \to Tid$ such that for each $j < p-1$ we have $(\overline{g}(j), \overline{ls}(j)[s(j)]) \to_{s(j)} (\overline{g}(j+1), \overline{ls}(j+1)[s(j)])$ and for all $t \in Tid \setminus \{s(j)\}$ we have $\overline{ls}(j)[t] = \overline{ls}(j+1)[t]$.

Let s be a schedule of an execution of length p of a multithreaded program and let $t \in Tid$. The *active set* of t in s is $s^{-1}(t) = \{j \mid s(j) = t\}$. The *number of phases* of t in s is the ordinal $N_t(s) := |s^{-1}(t)|$. The *phase position map* of t in s is $\beta_t(s, \cdot)\colon N_t(s) \to s^{-1}(t)$,

$$\beta_t(s,m) = \begin{cases} \min s^{-1}(t)\,, & \text{if } m = 0\,,\\ \min\{j > \beta_t(s,m-1) \mid s(j) = t\}\,, & \text{otherwise.} \end{cases}$$

The *phase number map* is $\delta_t(s, \cdot)\colon p \to 1 + N_t(s)$,

$$\delta_t(s,j) = \begin{cases} 0\,, \text{ if } j = 0\,,\\ \delta_t(s,j-1)\,, & \text{if } j > 0 \text{ and } s(j-1) \neq t\,,\\ \delta_t(s,j-1) + 1\,, & \text{if } j > 0 \text{ and } s(j-1) = t\,. \end{cases}$$

Informally, $\beta_t(s, \cdot)$ gives the position of a phase start by its number, and $\delta_t(s, \cdot)$ gives the number of the smallest phase that has not yet finished.

Proposition 3. *The phase number map counts the number of the finished phases and is inverse to the phase position map. Formally, let s be a schedule of an execution of length p and let $t \in Tid$. Then*
(a) for all positions $j < p$ we have $\delta_t(s,j) = |\{\tilde{j} < j \mid s(\tilde{j}) = t\}|$;
(b) for all active positions $j \in s^{-1}(t)$ we have $\beta_t(s, \delta_t(s,j)) = j$;
(c) for all phase numbers $m < N_t(s)$ we have $\delta_t(s, \beta_t(s,m)) = m$.

Proof. (a) For $j = 0$ we have $\delta_t(s,0) = 0 = |\emptyset| = |\{\tilde{j} < 0 \mid s(\tilde{j}) = t\}|$. If $j > 0$ and $s(j-1) \neq t$ then $\delta_t(s,j) = \delta_t(s,j-1) =$ [induction hypothesis] $|\{\tilde{j} < j-1 \mid s(\tilde{j}) = t\}| =$ [since $s(j-1) \neq t$] $|\{\tilde{j} < j \mid s(\tilde{j}) = t\}|$. If $j > 0$ and $s(j-1) = t$ then $\delta_t(s,j) = \delta_t(s,j-1) + 1 =$ [induction hypothesis] $|\{\tilde{j} < j-1 \mid s(\tilde{j}) = t\}| + 1 =$ [since $s(j-1) = t$] $|\{\tilde{j} < j \mid s(\tilde{j}) = t\}|$.

(b) We have to show that for any arbitrary fixed $t \in Tid$, a schedule s and $j \in s^{-1}(t)$ the number $\beta_t(s, \delta_t(s,j))$ is well-defined and equal to j. To show well-definedness, fix t and prove that $\delta_t(s,j) < N_t(s)$ for all $j \in s^{-1}(t)$ and all s by induction on the number of phases of t in s. If $N_t(s) = 0$ then $s^{-1}(t)$ is empty. Otherwise $N_t(s) = |\tilde{s}^{-1}(t) \dot\cup \{a\}|$ for some shorter schedule $\tilde{s}$ (of a prefix of an execution for s) and $a \in \operatorname{dom} s$ larger than any element of $\operatorname{dom} \tilde{s}$. Then $\delta_t(s,j) = \delta_t(\tilde{s},j) < N_t(\tilde{s}) < N_t(s)$ for all $j \in \tilde{s}^{-1}(t)$ and $\delta_t(s,a) \leq \delta_t(s,a-1) + 1 = \delta_t(\tilde{s},a-1) + 1 \leq$ [induction hypothesis] $N_t(\tilde{s}) + 1 = N_t(s)$. The equality is proven by fixing t and s and doing induction on j. If $j = \min s^{-1}(t)$ exists, then $\beta_t(s, \delta_t(s,j)) = \beta_t(s,0) = \min s^{-1}(t) = j$. Now let $s(j) = t$ and $j > \min s^{-1}(t)$. If $s(j-1) \neq t$, then $\beta_t(s, \delta_t(s,j)) = \beta_t(s,$

$\delta_t(s, \hat{j}+1))$ [where $\hat{j}$ is the largest element of $s^{-1}(t)$ less than j] $= \beta_t(s, \delta_t(s, \hat{j}) + 1) = \min\{\tilde{j} > \beta_t(s, \delta_t(s, \hat{j})) \mid s(\tilde{j}) = t\} =$ [induction hypothesis] $\min\{\tilde{j} > \hat{j} \mid s(\tilde{j}) = t\} = j$. If $s(j-1) = t$ then $\beta_t(s, \delta_t(s, j)) = \beta_t(s, \delta_t(s, j-1) + 1) = \min\{\tilde{j} > \beta_t(s, \delta_t(s, j-1)) \mid s(\tilde{j}) = t\} =$ [induction hypothesis] $\mid \{\tilde{j} > j - 1 \mid s(\tilde{j}) = t\} = j$.

(c) If $m = 0 < N_t(s)$ then $\delta_t(s, \beta_t(s, m)) = \delta_t(s, \min s^{-1}(t)) = 0$. Now let $0 < m < N_i(f)$. Then $\beta_t(s, m) > 0$. Notice that for all j such that $\beta_t(s, m-1) < j < \beta_t(s, m)$ we have $s(j) \neq t$, so for such j we have $\delta_t(s, j+1) = \delta_t(s, j)$. If any such j exists, then $\delta_t(s, \beta_t(s, m)) = \delta_t(s, j) = \delta_t(s, \beta_t(s, m-1) + 1)$. If no such j exists then $\beta_t(s, m) = \beta_t(s, m-1) + 1$ and so $\delta_t(s, \beta_t(s, m)) = \delta_t(s, \beta_t(s, m-1) + 1)$. In any case $\delta_t(s, \beta_t(s, m-1) + 1) =$ [since $s(\beta_t(s, m-1)) = t$] $\delta_t(s, \beta_t(s, m-1)) + 1 =$ [induction hypothesis] $(m-1) + 1 = m$.

$\square$

Restatement of Theorem 1. Let $P = (T_t)_{t \in Tid}$ and $P^{\#} = (T_t^{\#})_{t \in Tid}$ be multithreaded programs over the set *Global* of shared states such that $T_t^{\#}$ is an abstraction of T_t for all $t \in Tid$. Then, for all $g \in Global$, if P goes wrong from g, then $P^{\#}$ also goes wrong from g.

Proof. Write $T_t = (Global, Local, \to_t, Init_t, Wrong_t)$ and $T_t^{\#} = (Global, Local, \to_t^{\#}, Init_t^{\#}, Wrong_t^{\#})$. Let P go wrong from g. Then there is a nonempty execution (take a shortest one, i.e. with the least length) $e = (\overline{g}(j), \overline{ls}(j))_{j \leq k}$ such that $\overline{g}(0) = g$, $(g, \overline{ls}(0))$ is an initial state of P and $(\overline{g}(k), \overline{ls}(k)[t]) \in Wrong_t$ for some $t \in Tid$. Let s be a schedule of e. For each $t \in Tid$ there is an $N_t(s)$-phased execution

$$ep_t \ = \ ((\overline{g}(\beta_t(s, m)))_{m \leq N_t(s)}, (\overline{g}(\beta_t(s, m) + 1))_{m < N_t(s)}, (\overline{ls}(\beta_t(s, m))[t])_{m \leq N_t(s)})$$

where $\beta_t(s, N_t(s)) := \begin{cases} k\,, & \text{if } N_t(s) > 0\,, \\ 0\,, & \text{if } N_t(s) = 0\,. \end{cases}$ Since $T_t^{\#}$ is an abstraction of T_t, each such $N_t(s)$-phased execution ep_t is abstracted by an $N_t^{\#}$-phased execution

$$ep_t^{\#} \ = \ (\overline{g_t^{\#}}, \overline{g_t'^{\#}}, \overline{l_t^{\#}})\,,$$

where $\overline{g_t^{\#}}$ is a prefix of $(\overline{g}(\beta_t(s, m)))_{m \leq N_t(s)}$ and $\overline{g_t'^{\#}}$ is a prefix of $(\overline{g}(\beta_t(s, m) + 1))_{m < N_t(s)}$ $(t \in Tid)$. We construct an execution

$$e^{\#} = (\overline{\mathbf{g}^{\#}}(j), \overline{\mathbf{ls}^{\#}}(j))_{j \leq k}$$

of the composition of the abstract threads inductively such that $e^{\#}$ admits the same schedule s. Let $\overline{\mathbf{g}^{\#}}(0) = \overline{g}(0) = g$ and $\overline{\mathbf{ls}^{\#}}(0)[t] = \overline{l_t^{\#}}(\delta_t(s, 0)) = \overline{l_t^{\#}}(0)$ $(t \in Tid)$. Now assume that until some fixed $j < k$ the execution has been constructed such that for all $\tilde{j} \leq j$ we have $\overline{\mathbf{g}^{\#}}(\tilde{j}) = \overline{g}(\tilde{j})$ and $\overline{\mathbf{ls}^{\#}}(\tilde{j})[t] = \overline{l_t^{\#}}(\delta_t(s, \tilde{j}))$ $(t \in Tid)$. We construct the next state $(\overline{\mathbf{g}^{\#}}(j+1), \overline{\mathbf{ls}^{\#}}(j+1))$.

Assume that the abstract thread $T_{s(j)}^{\#}$ has made its last step in the constructed execution portion, i.e. that $\delta_{s(j)}(s, j) \geq N_{s(j)}^{\#}$. Then by definition of δ we have $N^{\#}_{s(j)} < N_{s(j)}(s)$. Thus the definition of the abstraction of an execution implies that $\mathit{Wrong}^{\#}_{s(j)} \ni \Big(\overline{g^{\#}_{s(j)}}(N^{\#}_{s(j)}), \overline{l^{\#}_{s(j)}}(N^{\#}_{s(j)})\Big) = \Big(\overline{g}\big(\beta_{s(j)}(s, N^{\#}_{s(j)})\big), \overline{l^{\#}_{s(j)}}(N^{\#}_{s(j)})\Big) =$ [for some active position $\tilde{j} \leq j$] $\Bigg(\overline{g}\Big(\beta_{s(j)}\big(s, \delta_{s(j)}(s, \tilde{j})\big)\Big), \overline{l^{\#}_{s(j)}}\big(\delta_{s(j)}(s, \tilde{j})\big)\Bigg) =$ [Prop. 3(b)] $\Big(\overline{g}(\tilde{j}), \overline{l^{\#}_{s(j)}}\big(\delta_{s(j)}(s, \tilde{j})\big)\Big) = \big(\overline{g}(\tilde{j}), \overline{\mathbf{ls}^{\#}}(\tilde{j})[s(j)]\big)$. Consequently, the abstract program $P^{\#}$ goes wrong from g, satisfying the theorem claim.

Otherwise the abstract thread $T^{\#}_{s(j)}$ has more steps left, i.e. $\delta_{s(j)}(s, j) < N^{\#}_{s(j)}$. We also have $\delta_i(s, j) \leq N^{\#}_t$ for all $t \in \mathit{Tid} \setminus \{s(j)\}$ (for $j = 0$ by definition of δ, for $j > 0$ from the fact that the position j of the execution is well-defined). Thus $\delta_{s(j)}(s, j+1) = \delta_{s(j)}(s, j) + 1 \leq N^{\#}_{s(j)}$ and for all $t \in \mathit{Tid} \setminus \{s(j)\}$ we have $\delta_t(s, j+1) = \delta_t(s, j) \leq N^{\#}_t$. Thus the new program state $(\overline{\mathbf{g}^{\#}}(j+1), \overline{\mathbf{ls}^{\#}}(j+1))$ $:= (\overline{g}(j{+}1), (\overline{l^{\#}_t}(\delta_t(s, j{+}1)))_{t \in \mathit{Tid}})$ is well-defined. To show that the j^{th} execution state is connected to the $(j+1)^{\text{st}}$ execution state by the transition relation of the abstract program, notice two facts:

- $\Big(\overline{\mathbf{g}^{\#}}(j), \overline{\mathbf{ls}^{\#}}(j)[s(j)]\Big) = \Big(\overline{g}(j), \overline{l^{\#}_{s(j)}}(\delta_{s(j)}(s, j))\Big)$ which by Prop. 3(b) equals $\Big(\overline{g}(\beta_{s(j)}(s, \delta_{s(j)}(s, j))), \overline{l^{\#}_{s(j)}}(\delta_{s(j)}(s, j))\Big) = \Big(\overline{g^{\#}_{s(j)}}(\delta_{s(j)}(s, j)), \overline{l^{\#}_{s(j)}}(\delta_{s(j)}(s, j))\Big)$ $\rightarrow_{s(j)} \Big(\overline{g'^{\#}_{s(j)}}(\delta_{s(j)}(s, j)), \overline{l^{\#}_{s(j)}}(\delta_{s(j)}(s, j) + 1)\Big)$ which by definition of δ equals $\Big(\overline{g'^{\#}_{s(j)}}(\delta_{s(j)}(s, j)), \overline{l^{\#}_{s(j)}}(\delta_{s(j)}(s, j{+}1))\Big)$ which by definition of $\overline{g'^{\#}_{s(j)}}$ is equal to $\Big(\overline{g}(\beta_{s(j)}(s, \delta_{s(j)}(s, j)) + 1), \overline{l^{\#}_{s(j)}}(\delta_{s(j)}(s, j + 1))\Big) =$ [using Proposition 3(b)] $\Big(\overline{g}(j + 1), \overline{l^{\#}_{s(j)}}(\delta_{s(j)}(s, j + 1))\Big) = \Big(\overline{\mathbf{g}^{\#}}(j + 1), \overline{\mathbf{ls}^{\#}}(j + 1)[s(j)]\Big)$.
- For $t \in \mathit{Tid} \setminus \{s(j)\}$ we have $\overline{\mathbf{ls}^{\#}}(j)[t] = \overline{l^{\#}_t}(\delta_t(s, j)) =$ [since $s(j) \neq t$] $\overline{l^{\#}_t}(\delta_t(s, j + 1)) = \overline{\mathbf{l}^{\#}}(j + 1)[t]$.

So $e^{\#}$ is constructed and it has the same length k and admits the same schedule s as e. Let $t \in \mathit{Tid}$ be the identifier of the failed thread, which by minimality of k implies $(\overline{g}(k), \overline{ls}(k)[t]) \in \mathit{Wrong}_t$. There are two cases

- $k = 0$. Then $N_t(s) = 0$ and $(\overline{g}(0), \overline{ls}(\beta_t(s, 0))[t]) \in \mathit{Wrong}_t$. Since $ep^{\#}$ is an abstraction of ep_t, we obtain $\mathit{Wrong}^{\#} \ni (\overline{g^{\#}_t}(0), \overline{l^{\#}_t}(0)) = (\overline{\mathbf{g}^{\#}}(0), \overline{\mathbf{ls}^{\#}}(0)[t])$.
- $k > 0$. The last shared-local pair of ep_t is $(\overline{g}(\beta_t(s, N_t(s))), \overline{ls}(\beta_t(s, N_t(s)))[t])$ $= (\overline{g}(k), \overline{ls}(k)[t]) \in \mathit{Wrong}_t$. Since $ep^{\#}_t$ abstracts ep_t, we have $\mathit{Wrong}^{\#}_t \ni$ $(\overline{g^{\#}_t}(N^{\#}_t), \overline{l^{\#}_t}(N^{\#}_t)) =$ [due to $N^{\#}_t = N_t(s)$] $(\overline{g}(\beta_t(s, N_t(s))), \overline{l^{\#}_t}(N_t(s))) =$ [def. of $\overline{\mathbf{g}^{\#}}$ and monotonicity and surjectivity of $\delta_t(s, \cdot)$] $(\overline{\mathbf{g}^{\#}}(k), \overline{l^{\#}_t}(\delta_t(s, k)))$ $= (\overline{\mathbf{g}^{\#}}(k), \overline{\mathbf{ls}^{\#}}(k)[t])$.

In each case $P^{\#}$ goes wrong via a failing execution $e^{\#}$. □

The definition of abstraction implies directly that each thread is an abstraction of itself:

Theorem 4 (Completeness). *For each correct multithreaded program P there is a correct multithreaded program $P^{\#}$ consisting of abstractions of threads of P.*

Restatement of Theorem 2. Let $T = (\mathit{Global}, \mathit{Local}, \rightarrow, \mathit{Init}, \mathit{Wrong})$ and $T^{\#} = (\mathit{Global}, \mathit{Local}^{\#}, \rightarrow^{\#}, \mathit{Init}^{\#}, \mathit{Wrong}^{\#})$ be threads over a nonempty set of shared states Global. Then $T^{\#}$ is an abstraction of T if and only if the conformance checker $C(T, T^{\#})$ is correct.

Proof. "$\Rightarrow$": Otherwise there is a failing execution (choose a shortest one) $e = (\bar{l}_j, \overline{F}_j)_{j \leq p}$ of the conformance checker.

Case $p = 0$. Either $\mathit{Init}^{\#} = \emptyset$ or $W(\bar{l}_0) \not\subseteq W^{\#}(\mathit{Init}^{\#})$. Since e is a nonempty sequence, $\mathit{Start} \neq \emptyset$, so $\mathit{Init} \neq \emptyset$, so a phased execution of T exists, namely $ep = (\{(0, g)\}, \emptyset, \{(0, \bar{l}_0)\})$ for some $g \in \mathit{Global}$. It is abstracted by some phased execution $ep^{\#}$ of $T^{\#}$ by the theorem assumption, so $\mathit{Init}^{\#} \neq \emptyset$. Thus $W(\bar{l}_0) \not\subseteq W^{\#}(\mathit{Init}^{\#})$. So there is some $g \in \mathit{Global}$ such that $(g, \bar{l}_0) \in \mathit{Wrong}$ but for all $l^{\#} \in \mathit{Init}^{\#}$ we have $(g, l^{\#}) \notin \mathit{Wrong}^{\#}$. On the other hand, since ep is abstracted by $ep^{\#}$, there is some $l^{\#} \in \mathit{Init}^{\#}$ with $(g, l^{\#}) \in \mathit{Wrong}^{\#}$. A contradiction!

Case $p > 0$. Then there are p-long sequences $\overline{g}$, $\overline{g'}$ of shared states such that for all $j < p$ we have

$$\overline{g}_j \notin W^{\#}(\overline{F}_j) \text{ and } (\overline{g}_j, \bar{l}_j) \rightarrow (\overline{g'_j}, \bar{l}_{j+1}) \tag{1}$$

$$\text{and } \overline{F}_{j+1} \; = \; \{m' \mid \exists\, m \in \overline{F}_j : (\overline{g}_j, m) \rightarrow^{\#} (\overline{g'_j}, m')\}\,. \tag{2}$$

Extend $\overline{g}$ by setting $\overline{g}_p \in \begin{cases} W(\bar{l}_p) \setminus W^{\#}(\overline{F}_p)\,, & \text{if } W(\bar{l}_p) \not\subseteq W^{\#}(\overline{F}_p)\,, \\ \mathit{Global}\,, & \text{otherwise.} \end{cases}$

Then $ep := (\overline{g}, \overline{g'}, \bar{l})$ is a p-phased execution of T. It is abstracted by some $p^{\#}$-phased execution $ep^{\#} = ((\overline{g}_j)_{j \leq p^{\#}}, (\overline{g'_j})_{j < p^{\#}}, \overline{l^{\#}})$ of $T^{\#}$.

We claim:

$$\overline{l^{\#}_j} \in \overline{F}_j \text{ for all } j \leq p^{\#}\,. \tag{3}$$

Indeed, $\overline{l^{\#}_0} \in \mathit{Init}^{\#} = \overline{F}_0$; the fact that $ep^{\#}$ is a phased execution implies $(\overline{g}_j, \overline{l^{\#}_j}) \rightarrow^{\#} (\overline{g'_j}, \overline{l^{\#}_{j+1}})$ for all $j < p^{\#}$ and so if $\overline{l^{\#}_j} \in \overline{F}_j$ for some $j < p^{\#}$ then (2) implies $\overline{l^{\#}_{j+1}} \in \overline{F}_{j+1}$. The claim is proven.

Case $\overline{F}_p = \emptyset$. The claim (3) implies $\overline{F}_j \neq \emptyset$ for all $j \leq p^{\#}$. Since $\overline{F}_p = \emptyset$, we obtain $p^{\#} < p$. Since $ep^{\#}$ abstracts ep, we obtain $(\overline{g}_{p^{\#}}, \overline{l^{\#}_{p^{\#}}}) \in \mathit{Wrong}^{\#}$.

However $p^{\#} < p$ and (1) imply that $\overline{g}_{p^{\#}} \notin W^{\#}(\overline{F}_{p^{\#}})$. Thus $\overline{l^{\#}_{p^{\#}}} \in \overline{F}_{p^{\#}}$ implies $\overline{g}_{p^{\#}} \notin W^{\#}(\overline{l^{\#}_{p^{\#}}})$. So $(\overline{g}_{p^{\#}}, \overline{l^{\#}_{p^{\#}}}) \notin \mathit{Wrong}^{\#}$. A contradiction!

Case $W(\bar{l}_p) \not\subseteq W^{\#}(\overline{F}_p)$.

Case $p^\# < p$. Since $ep^\#$ abstracts ep we have $(\overline{g}_{p^\#}, \overline{l^\#_{p^\#}}) \in \mathit{Wrong}^\#$. Thus $\overline{g}_{p^\#} \in W^\#(\overline{l^\#_{p^\#}})$. Combining (1) and (3) we obtain $\overline{g}_{p^\#} \notin W^\#(\overline{l^\#_{p^\#}})$. A contradiction!

Case $p^\# = p$. Since $\overline{g}_p \in W(\overline{l}_p)$, we have $(\overline{g}_p, \overline{l}_p) \in \mathit{Wrong}$. Since $ep^\#$ abstracts ep we have $(\overline{g}_p, \overline{l^\#_p}) \in \mathit{Wrong}^\#$. Since $\overline{g}_p \notin W^\#(\overline{F}_p)$ and since (3) implies $\overline{l^\#_{p^\#}} = \overline{l^\#_p} \in \overline{F}_p$, we have $\overline{g}_p \notin W^\#(\overline{l^\#_p})$, and so $(\overline{g}_p, \overline{l^\#_p}) \notin \mathit{Wrong}^\#$. A contradiction!

"$\Leftarrow$": Otherwise there is some phased execution (take a shortest one) $ep = ((\overline{g}_j)_{j\leq p}, (\overline{g'_j})_{j<p}, (\overline{l}_j)_{j\leq p})$ of T such that no phased execution of $T^\#$ abstracts ep. Define the sequence $\overline{F}$ of length $p+1$ inductively by letting $\overline{F}_0 = \mathit{Init}^\#$ and $\overline{F}_{j+1} = \{m' \mid \exists\, m \in \overline{F}_j : (\overline{g}_j, m) \rightarrow^\# (\overline{g'_j}, m')\}$ for $j < p$. Now we show that $\overline{e} = (\overline{l}_j, \overline{F}_j)_{j\leq p}$ is an execution of the conformance checker.

First notice that $\overline{l}_0 \in \mathit{Init}$ (since ep is a phased execution) and $\overline{F}_0 = \mathit{Init}^\#$ by definition. Now assume that we have proven that $(\overline{e}_{\tilde{j}})_{\tilde{j}\leq j}$ is an execution of the conformance checker for some $j < p$ and we are going to show that $(\overline{e}_{\tilde{j}})_{\tilde{j}\leq j+1}$ is an execution of the conformance checker. It suffices to show that $(\overline{l}_j, \overline{F}_j) \rightsquigarrow (\overline{l}_{j+1}, \overline{F}_{j+1})$. We show:

- $(\overline{g}_j, \overline{l}_j) \rightarrow (\overline{g'_j}, \overline{l}_{j+1})$. Follows from the fact that ep is a phased execution.
- $\overline{g}_j \notin W^\#(\overline{F}_j)$. For the purpose of contradiction assume the contrary, i.e. that there is some $l^\# \in \overline{F}_j$ such that $\overline{g}_j \in W^\#(l^\#)$. Then $(\overline{g}_j, l^\#) \in \mathit{Wrong}^\#$. Construct the sequence $(\overline{l^\#_{\tilde{j}}})_{\tilde{j}\leq j}$ inductively backwards by setting $\overline{l^\#_j} := l^\#$ and $\overline{l^\#_{\tilde{j}-1}}$ as any element of $\overline{F}_{\tilde{j}-1}$ such that $(\overline{g}_{\tilde{j}-1}, \overline{l^\#_{\tilde{j}-1}}) \rightarrow^\# (\overline{g'_{\tilde{j}-1}}, \overline{l^\#_{\tilde{j}}})$ (such an element exists by definition of $\overline{F}_{\tilde{j}}$) for all $\tilde{j} > 0$ with $\tilde{j} \leq j$. Notice that $\overline{l^\#_0} \in \mathit{Init}^\#$. So $ep^\# = ((\overline{g}_{\tilde{j}})_{\tilde{j}\leq j}, (\overline{g'_{\tilde{j}}})_{\tilde{j}<j}, (\overline{l^\#_{\tilde{j}}})_{\tilde{j}\leq j})$ is a phased execution of $T^\#$. Then $(\overline{g}_j, \overline{l^\#_j}) \in \mathit{Wrong}^\#$ implies that $ep^\#$ abstracts ep. A contradiction!
- $\overline{F}_{j+1} = \{m' \mid \exists\, m \in \overline{F}_j : (\overline{g}_j, m) \rightarrow^\# (\overline{g'_j}, m')\}$ by definition.

By theorem assumption, $\overline{e}$ is not failing. Thus for all $j \leq p$ we have $\overline{F}_j \neq \emptyset$ and $W(\overline{l}_j) \subseteq W^\#(\overline{F}_j)$. Distinguish two cases.

- There is some $j \leq p$ such that $(\overline{g}_j, \overline{l}_j) \in \mathit{Wrong}$. Then $\overline{g}_j \in W(\overline{l}_j)$, so $\overline{g}_j \in W^\#(\overline{F}_j)$, thus there is some $l^\# \in \overline{F}_j$ with $(\overline{g}_j, l^\#) \in \mathit{Wrong}^\#$. As before, we construct the sequence $(\overline{l^\#_{\tilde{j}}})_{\tilde{j}\leq j}$ inductively backwards by setting $\overline{l^\#_j} := l^\#$ and $\overline{l^\#_{\tilde{j}-1}}$ as any element of $\overline{F}_{\tilde{j}-1}$ such that $(\overline{g}_{\tilde{j}-1}, \overline{l^\#_{\tilde{j}-1}}) \rightarrow^\# (\overline{g'_{\tilde{j}-1}}, \overline{l^\#_{\tilde{j}}})$ (such an element exists by definition of $\overline{F}_{\tilde{j}}$) for all $\tilde{j} > 0$ with $\tilde{j} \leq j$. Notice that $\overline{l^\#_0} \in \mathit{Init}^\#$. So $ep^\# = ((\overline{g}_{\tilde{j}})_{\tilde{j}\leq j}, (\overline{g'_{\tilde{j}}})_{\tilde{j}<j}, (\overline{l^\#_{\tilde{j}}})_{\tilde{j}\leq j})$ is a phased execution of $T^\#$. Then $(\overline{g}_j, \overline{l^\#_j}) \in \mathit{Wrong}^\#$ implies that $ep^\#$ abstracts ep. A contradiction!

– For all $j \leq p$ we have $(\overline{g}_j, \overline{l}_j) \notin \mathit{Wrong}$. As before, we construct the sequence $(\overline{l^{\#}_{\tilde{j}}})_{\tilde{j} \leq p}$ inductively backwards by setting $\overline{l^{\#}_p}$ to be any element of $\overline{F}_p \neq \emptyset$ and $\overline{l^{\#}_{\tilde{j}-1}}$ as any element of $\overline{F}_{\tilde{j}-1}$ such that $(\overline{g}_{\tilde{j}-1}, \overline{l^{\#}_{\tilde{j}-1}}) \rightarrow^{\#} (\overline{g'_{\tilde{j}-1}}, \overline{l^{\#}_{\tilde{j}}})$ (such an element exists by definition of $\overline{F}_{\tilde{j}}$) for all $\tilde{j} > 0$ with $\tilde{j} \leq p$. Notice that $\overline{l^{\#}_0} \in \mathit{Init}^{\#}$. So $ep^{\#} = ((\overline{g}_{\tilde{j}})_{\tilde{j} \leq p}, (\overline{g'_{\tilde{j}}})_{\tilde{j} < p}, (\overline{l^{\#}_{\tilde{j}}})_{\tilde{j} \leq p})$ is a phased execution of $T^{\#}$. Then $(\overline{g}_p, \overline{l}_p) \notin \mathit{Wrong}$ implies that $ep^{\#}$ abstracts ep. A contradiction!

□