Journal Of Automated Reasoning manuscript No.
(will be inserted by the editor)

On Automation in the Verification of Software Barriers: Experience Report

Alexander Malkis · Anindya Banerjee

accepted on 14 April 2013, published on 20 September 2013, last updated on 13 September 2021

Abstract We present an experience report on automating the verification of the software barrier synchronization primitive. The informal specification of the primitive is: when a thread calls the software barrier function, the thread halts until all other threads call their instances of the software barrier function. A successful software barrier call ensures that each thread has finished its portion of work before the threads start exchanging the results of these portions of work. While software barriers are widely used in parallel versions of major numerical algorithms and are indispensable in scientific computing, software barrier algorithms and their implementations scarcely have been verified. We improve the state of the art in proving the correctness of the major software barrier algorithms with off-the-shelf automatic verification systems such as Jahob, VCC, Boogie, Spin and Checkfence. We verify a central barrier, a C implementation of a barrier, a static tree barrier, a combining tree barrier, a dissemination barrier, a tournament barrier, a barrier with its client and a barrier on a weak memory model. In the process, we introduce a novel theorem proving method for proving validity of formulas containing cardinalities of comprehensions and improve the capabilities of one of the verification systems. Based on our experience, we propose new challenges in the verification of software barriers.

Keywords barrier · verification · invariant · safety · verifier · automation

CR Subject Classification D.2.4 · F.3.1 · D.4.1

Alexander Malkis
Institut für Informatik (I4)
Technische Universität München
Boltzmannstr. 3
85748 Garching bei München, Germany
Tel.: (+49) 89 289 17337
Fax: (+49) 89 289 18579
E-mail: malkis@sec.in.tum.de

Anindya Banerjee
IMDEA Software Institute
Edificio IMDEA Software
Campus Montegancedo UPM
28223-Pozuelo de Alarcón, Madrid, Spain
E-mail: Anindya.Banerjee@imdea.org

1 Introduction

We verify implementations of the software barrier, which is a standard concurrency primitive that enables threads to synchronize in the following manner. Namely, if a thread calls the software barrier function, the barrier function starts waiting until all other threads call their instances of the barrier function. After all other threads have called the barrier function, the waiting stops, and all the threads are allowed to proceed with the instruction following the call to the barrier function.

Software barriers are available in most libraries, including PThreads, OpenMP, MPI, CUDA, java.util.concurrent and .NET. The barrier algorithms range from rich-feature ones to high-performance ones. The barrier primitive is routinely used in shared-memory programs. We will examine the spread of the barriers later in Section 9, for now demonstrating a typical usage of the barrier by the following code snippet:

```
                        Initially c=0;
// One thread.                  || // Another thread.
c++; // Atomic increment.       || c++; // Atomic increment.
barrier();                      || barrier();
... // Here rely on c=2.        || ... // Here rely on c=2.
```

How do we know whether the used barrier implementations are really correct? After all, errors in concurrent algorithms are common even after reviewed publications. We are the first, to the best of our knowledge, to systematically expound the proofs of correctness of the major software barriers in off-the-shelf automatic verification tools. Our contribution is not only to prove the correctness of the barriers, but also to improve one of the verification tools, Jahob [27]. The tool improvements concern theorem proving: now Jahob can handle more and larger formulas than it could handle before, implying that now Jahob is able to prove stronger properties of programs. In particular, we have implemented a new theorem proving method, called *free comprehension-cardinality instantiation*. This and other improvements are going to be described during exposition; they are also incorporated into the development trunk of Jahob.

The *barrier property* we would like to verify is: if one thread passes the barrier, all the other threads have already arrived at it. Our key contribution is proving just this fact for the major barrier algorithms.

Our priority in this exposition is to handle a large number of different barriers: to keep the presentation simple, for most algorithms we prove the barrier property just for the first call of each thread. Other issues like handling a large number of control flow locations are inherent also to sequential programs and can be dealt with by appropriate methods [15].

For most of the barriers, we follow a standard verification process [33] in which a program is annotated with an inductive invariant up front. Different algorithms admit a different amount of proof automation in this respect, reflecting the state of the art:

- Some will just need support in proving that a user-given invariant is indeed inductive.
- Some will require only an inductive invariant, the inductiveness proof is fully automatic.
- Some will be fully automatic, not requiring even an inductive invariant for proving the barrier property.

We verify most of the algorithms assuming sequentially consistent memory and unbounded integers, revealing a gap between real hardware and available verification tools. If a program is verified against sequential consistency and unbounded integers, without special precautions the program would in general not run correctly on extant hardware. But verification against the ideal computation model is still of great use: almost every industrially relevant application, including the software barriers, can be potentially adapted to real hardware, e.g. by inserting overflow checks and instructions that flush hardware caches. In addition to the verification against sequential consistency, we have advanced one step further by verifying one barrier algorithm against a weak memory model. Integer overflows are not a problem for us: in all cases but one, the used integer interval is $[0, \text{number of threads}]$, and in the exceptional case there is a time counting variable that, even in the worst scenario, would not overflow within 64 years, thus being practically benign.

All the experiments are fully reproducible. Our online material [32] provides the entire code as well as the instructions for reproducing verification.

Our *contributions* are:

- discovery of inductive invariants of major barrier algorithms for sequential consistency,
- verification of major barrier algorithms for sequential consistency in automatic verifiers,
- verification of the central barrier algorithm for weak memory in an automatic verifier,
- a new proof method for formulas exhibiting cardinalities of comprehensions,
- identifying a sufficient toolbox of reasoning techniques for barriers,
- significant improvements to the Jahob verification system,
- bug reporting or bug fixing in the other verification tools Z3, VCC, Checkfence,
- identification of future challenges in the area of barrier verification.

2 Overview

This section outlines the organization of the barrier proofs.

For sequentially consistent memory, we will consider several barrier algorithms, a barrier implementation in multithreaded C and a barrier with a client; for a weak memory model we will look at a central software barrier. We will describe each of these benchmarks in the following 6 stages.

I. Algorithm.
We examine a barrier algorithm at hand. Usually it is given by pseudocode or by code in a multithreaded version of some imperative programming language. In some cases we remove or rewrite parts of the code which are only weakly related to the aforementioned barrier property and at the same time make verification intractable (e.g. features such as using the barrier more than once, running external code, error processing, adding/removing participants, etc. are less relevant). In any case we retain the parts of the code that are central to the barrier property.

II. Required reasoning.
After settling on the algorithm we determine what kind of reasoning is required to prove the barrier property. Various algorithms require reasoning about integers, sets with cardinalities, two-dimensional arrays, maps or trees. Determining the required reasoning influences the later choice of the tool to verify the algorithm.

III. High-level proof.
If the reasoning requires an inductive invariant, we either manually produce an inductive invariant that implies the barrier property or say how the invariant is automatically derived. If the required reasoning does not involve an inductive invariant, we describe the underlying proof structure instead.

IV. Encoded proof.
After obtaining a program and the inductive property we encode them into the tool that has the best support for the required reasoning principles. The Boogie tool [29] can reason very well about arrays, the Jahob tool [27] has good decision procedures for trees and sets; verification of real C implementations is best handled in VCC [41], programs on bounded integers can be handed over to the finite-state model checker Spin [22], and programs on weak memory can be handled by Checkfence [9].

V. Using the tool.
The use of a verifier resembles the use of a command-line compiler: if the first attempt fails, we repair the input to the verifier (in some cases we repair the verifier) and restart the verifier. After some iterations verification succeeds; in the presentation the results of the last successful attempts will be shown.

A verification attempt might fail for different reasons, e.g. the tool should prove the absence of null-pointer dereferences but cannot prove that. Then we improve either the program (e.g. insert a null-pointer check into the code) or the inductive property to be proven (e.g. insert a conjunct stating that a particular pointer is never null) or the capabilities of the tool (e.g. if the tool fails in calling a theorem prover that would prove the absence of null-pointer dereferences, we repair the tool).

VI. Beyond the frontier.
Finally, we show a slight change in the barrier algorithm or in the property to be shown that still prevents automatic verification with the latest technology. Such changed versions provide future challenges in verification of concurrent programs.

3 Caveats

Our experience in the verification of software barriers exposed certain deficiencies and limitations of modern verification technology which we would like to state up front.

First, some programs can be verified for an unbounded number of threads, some others cannot, while still others do not have a meaningful parameterized description. In certain cases the verification process will require a bounded number of threads, while other quantities may still stay unbounded.

Next, for most programs we verify the barrier property just for the first call. This simple property certainly builds a basis for stronger and more elaborate properties. In most cases, an extension to more than one call would be theoretically easy: usually it would require just a larger program and a more complicated property to prove, and the reasoning complexity would increase only moderately. However, practically, even slightly larger programs or properties possibly expressed in the same logic would immediately cross the automation frontier: at the time the research was conducted, virtually all the used tool chains had problems with the relatively small formulas and relatively small state spaces arising from verification of one-time barriers.

Moreover, existing automatic verification tool chains do not admit compositional verification of both the barrier and its clients in the same framework in the following sense. Ideally, one would first verify a property of the given barrier implementation. Second, one would automatically use this property in proving high-level properties of the code of the clients without reexamining the barrier implementation. However, in this paper, we seamlessly verify one implementation of a barrier with its client.

Furthermore, the used research tool chains do not implement all of the mechanization that would be required to verify a barrier with a single push of a button. For example, automatic syntactic translation from a multithreaded program into a nondeterministic sequential one for Jahob and Boogie is missing. Such well understood, but so far absent features can be implemented on need.

Finally, one researcher on barriers surmised that "a full formal correctness proof for these codes would be a lengthy and boring exercise" [31]. We show, to the contrary, that it is possible to delegate the long and boring part of theorem proving to automatic verification tools.

4 Central barriers

This section describes verification of the central barrier. We will start with the simplest algorithm, then proceed with an implementation in C and finish with a comparison of the two. As we will see, even on the simplest example of a barrier, the key challenge emerges immediately: how to reason about cardinalities of comprehensions over program locations. We will discuss how to solve this problem.

4.1 The simplest central barrier

I. Algorithm.
A trivial implementation of the barrier involves a counter. The counter is initially the number of (participating) threads and gets decreased when a thread enters the barrier. Only when the counter is zero are all threads allowed to proceed (in the sequel, all the code will be displayed in small letters):

```
shared int count; // initially the number of participating threads
// The function to be called by each thread:
void barrier() {
  // Decrement the number of threads that have not yet called the barrier:
  count−−; // the current thread has called the barrier.
  while(count!=0); // the current thread waits until all threads have joined.
}
```

After the counter reaches zero, it is never reset – this barrier is for one-time use only. The program which we analyze is the parallel composition of the threads that execute the function body; i.e., each thread executes the following code:

```
A: count−−;     B: while(count≠0);     C:
```

II. Required reasoning.
First we will delineate the problem that arises when trying to reason about the program; then we will suggest a few solutions to this problem.
II.(i) Problem description.
We want to show the barrier property: when one thread has arrived at location `C`, all the others are no longer at location `A`. To show the property, we need the ability to express that at any time point in the execution of the program, the value of the program variable `count` is at least the number of threads at location `A`. (When a variable identifier acts as a program variable, we write it as code, i.e. in small letters; when the identifier acts as a logical variable, we write it in normal font.) We also need the ability to say that if

some thread is at location $\texttt{C}$, then $\mathsf{count} = 0$. The number of threads at location $\texttt{A}$ is the cardinality of $\{t \in \mathit{Tid} \mid pc_t = \texttt{A}\}$, where *Tid* is the set of thread identifiers and pc_t is the program counter of thread t. The formula

$$J \quad = \quad (\mathsf{count} \geq |\{t \in \mathit{Tid} \mid pc_t = \texttt{A}\}| \ \wedge \ \forall t \in \mathit{Tid} \colon (pc_t = \texttt{C} \Rightarrow \mathsf{count} = 0))$$

is an inductive invariant which implies the barrier property: if some thread t satisfies $pc_t = \texttt{C}$, then the first conjunct yields $0 \geq |\{t \in \mathit{Tid} \mid pc_t = \texttt{A}\}|$, so there is no thread t with $pc_t = \texttt{A}$.

To prove inductiveness of J, it suffices to show two facts.

Initial condition: J holds initially, i.e. when all the threads are at location $\texttt{A}$ and count is the number of all threads,

Stability: J implies the weakest precondition of J with respect to each of the two transitions $\texttt{A} \to \texttt{B}$ and $\texttt{B} \to \texttt{C}$ of an arbitrary thread.

Each of the proof obligations is an implication where J (in case we check the initial condition) or its weakest precondition (in case we check stability) is in the consequent. Since J is a conjunction and weakest precondition distributes over conjunctions, each occurring consequent is a conjunction. Each proof obligation is proven by splitting the consequent into two conjuncts and proving each of the conjuncts separately, which results in two checks for the initial condition and four checks for the stability, a total of six checks. The resulting formulas, whose validity has to be shown, exhibit several complications at the same time: unbounded integers (count), quantifiers ($\forall t$), finite sets (*Tid*), comprehensions ($\{\ldots\}$) and cardinalities ($|\ldots|$). As of spring of 2012, no usable general-purpose automatic theorem prover can reason about such formulas. (We have even tried an interactive theorem prover, Isabelle—the smallest proof of the hardest formula requires a manually typed three-lines-long lemma.)

II.(ii) Solution: using counter abstraction.

One known way to circumvent these difficulties is to use counter abstraction, i.e. introduce an auxiliary variable for each control flow location that counts the number of threads at that location. We obtain, e.g. a variable *no_of_threads_at_location_A*, for which the invariant $\mathit{no\_of\_threads\_at\_location\_A} = |\{t \in \mathit{Tid} \mid pc_t = \texttt{A}\}|$ holds by construction and the invariant $\mathit{no\_of\_threads\_at\_location\_A} = \mathsf{count}$ holds, since both variables in the equality are decremented simultaneously. In experiments with counter abstraction the program can be automatically verified within a second.

The method of counter abstraction is easy for toy programs like this one. Unfortunately, it also breaks easily when the comprehension under the cardinality is syntactically different from "the number of threads at a single fixed location". Prior syntactic transformations, e.g. rewriting a multithreaded program into a nondeterministic sequential one, usually render counter abstraction useless. Also, real-life central barriers as in Section 7 require slightly different comprehensions in the invariant (e.g. "the number of threads t in an interval of locations such that a local Boolean flag of t is raised")—which also makes counter abstraction useless, at least in the strict way it is defined. The question arises whether counter abstraction can be made more flexible, and whether such flexibility could be reached automatically. As we will see, our approach, in a broad sense, will subsume such flexible extensions. We now demonstrate two new methods that relax or even completely get rid of the restriction of auxiliary integer variables with a fixed meaning (e.g. the meaning of a variable like *no_of_threads_at_location_A* does not depend on what the original program is doing).

II.(iii) Solution: using BAPA directly.

In the *first* method, we convert the program and the invariant such that the proof obligations lie in BAPA, the logic of Boolean Algebra and Presburger Arithmetic with cardinality constraints.

In short, BAPA allows us to express sets, their unions and intersections, their cardinalities and basic arithmetic over them, but does not allow to reason about the membership relation "$\in$". More concretely, one can use set variables A, B, C, $\ldots$, expressions like $|(A \cap B) \cup C|$, but the syntax phrase $\{t \in \mathit{Tid} \mid pc_t = \mathtt{A}\}$, appearing in J, cannot be expressed in BAPA. However, we could give this phrase a name, say, A and treat it as an object without internal structure. Similarly, we use B for $\{t \in \mathit{Tid} \mid pc_t = \mathtt{B}\}$ and C for $\{t \in \mathit{Tid} \mid pc_t = \mathtt{C}\}$. We rewrite the multithreaded program such that it operates with A, B, C to reflect the change of the program counter. Additionally, the variable *Tid* will represent the set of all threads in the system. (We will see the details of the rewritten code later on page 10.) One can view the variable *no_of_threads_at_location_A* from the counter abstraction method, roughly speaking, as an abstraction of the set variable A. In this new method, we obtain the ability to express the notion of different threads and reason about them in addition to reasoning about just their numbers. Thus, our method has a wider applicability in comparison to the counter abstraction.

II.(iv) Solution: using AUFLIA and BAPA.

In the *second* method, we leave the program and the proof obligation as they are, neither adding any auxiliary variables nor applying any abstraction, thus without incurring upfront costs or an upfront loss of precision. Instead, we discharge the resulting higher-order verification conditions by transforming them into a first-order logic, thus delaying the loss of precision until the semantics of the program gets visible in proof obligations.

Preparing to show the transformation, we start with a simple logical result.

Proposition 1 *Let ψ and θ be higher-order-logic terms of Boolean type. Let σ be a type-preserving substitution of variables by terms. Assume that*
(1) $\sigma(\psi) \vee \sigma(\theta)$ is valid and
(2) $\psi \vee \neg\theta$ is valid.
Then $\sigma(\psi)$ is valid.

Proof. For an arbitrary type-correct valuation of variables we have $\sigma(\psi) \Leftrightarrow \sigma(\psi \vee \mathit{false}) \Leftrightarrow \sigma(\psi \vee (\theta \wedge \neg\theta)) \Leftrightarrow \sigma((\psi \vee \theta) \wedge (\psi \vee \neg\theta)) = (\sigma(\psi) \vee \sigma(\theta)) \wedge \sigma(\psi \vee \neg\theta) \Leftrightarrow \mathit{true} \wedge \sigma(\mathit{true}) \Leftrightarrow \mathit{true}$. □

To show the transformation, assume that we have to discharge a verification condition, which is a formula, say, ϕ, in higher-order logic. The free variables of ϕ correspond to the program variables and the data structures in the following states.

- An initial state, namely when checking that the initial condition implies the inductive invariant or when checking the input-output relation.
- The state before some transition, namely when checking that an inductive invariant is stable under the transition.
- The state after this transition; but if the transition is at most boundedly nondeterministic, logical variables corresponding to the state after the transition can be eliminated.

The formula ϕ shall be proven valid.

The following simplified example of such a formula states that the first conjunct of J holds after an arbitrary thread, say, t_1, takes a step from $\mathtt{A}$ to $\mathtt{B}$:

$$\phi \quad = \quad \begin{pmatrix} (|\{t \mid t \in \mathit{Tid} \wedge pc_t = \mathtt{A}\}| \leq \mathsf{count} \ \wedge \ t_1 \in \mathit{Tid} \ \wedge \ pc_{t_1} = \mathtt{A} \ \wedge \ \text{other conjuncts}) \\ \Rightarrow \ |\{t \mid t \in \mathit{Tid} \ \wedge \ (pc[t_1 \mapsto \mathtt{B}])_t = \mathtt{A}\}| \leq \mathsf{count} - 1 \end{pmatrix}.$$

Henceforth, “other conjuncts” will contain irrelevant formulas not influencing the subsequent treatment. (Here, for instance, “other conjuncts” contain the second conjunct of J.) Also notice that the logical variables specifying the values of count and pc after the transition have already been substituted by $\mathsf{count}-1$ and $pc[t_1 \mapsto \mathtt{B}]$, respectively.

We wish to prove validity of ϕ, but cardinalities of comprehensions (whose semantics is in this case necessary to establish the validity) cannot be directly treated by current theorem provers. Our remedy is constructing formulas ψ and θ as well as a substitution σ such that

– the preconditions of Prop. 1 are met,
– $\phi = \sigma(\psi)$, and
– the formulas $\sigma(\psi) \vee \sigma(\theta)$ and $\psi \vee \neg\theta$ have fewer cardinalities of comprehensions than ϕ and are provably valid.

To construct ψ, we introduce fresh logical variables for the comprehensions under cardinalities. To illustrate this point, if $\bar{p}_t$, $\hat{p}_t$ and $\check{p}_t$ are the formulas describing the program counter of a thread t in the initial state, before the transition and after the transition, then the formulas $|\{t \mid t \in \mathit{Tid} \wedge \bar{p}_t = \mathtt{A}\}|$, $|\{t \mid t \in \mathit{Tid} \wedge \hat{p}_t = \mathtt{A}\}|$ and $|\{t \mid t \in \mathit{Tid} \wedge \check{p}_t = \mathtt{A}\}|$ get substituted by cardinalities of fresh variables, say, $|\bar{A}|$, $|\hat{A}|$ and $|\check{A}|$. Syntactically equal comprehensions get substituted by equal variables. The inverse substitution, which maps these fresh variables to the corresponding comprehensions, will be σ. In all our verification conditions only unbounded comprehensions occur. Such comprehensions are outside variable binders, i.e. outside quantifiers, lambda abstractions, other comprehensions. Were a comprehension to occur under a binder, we would leave it unabstracted at this stage. This substitution process produces a formula ψ whose validity implies the validity of ϕ.

For our running example,

$$\psi \quad = \quad \big(\big(|\hat{A}| \le \mathsf{count} \ \wedge\ t_1 \in \mathit{Tid} \ \wedge\ pc_{t_1} = \mathtt{A} \ \wedge\ \text{other conjuncts}\big) \ \Rightarrow\ |\check{A}| \le \mathsf{count}-1\big)\,.$$

Though ψ lies in first-order logic for all our verification conditions, ψ is not yet useful by itself because it is too strong to be valid.

Now we construct a first-order-logic formula θ, which will be a conjunction of several atomic formulas, by checking

– membership relations between already present free variables and introduced fresh variables (e.g. whether $t_1 \in \hat{A}$ or $t_1 \notin \hat{A}$) and
– subset relations between introduced fresh variables (e.g. whether $\hat{A} \subseteq \bar{A}$ or $\hat{A} \not\subseteq \bar{A}$)

under the assumption $\neg\phi$ (i.e. assuming $\neg\sigma(\psi)$). While checking such an auxiliary membership or subset relation we have full access to the definitions of set comprehensions, and such a check can be executed by any suitable theorem prover. In all our examples the check does not rely on cardinalities of comprehensions, so we use an SMTLIB solver for AUFLIA, the logic of Arrays, Uninterpreted Functions and Linear Integer Arithmetic. Thus, if $\phi \vee (t_1 \in \mathit{Tid} \wedge \hat{p}_{t_1} = \mathtt{A})$ is valid, we make $t_1 \in \hat{A}$ a conjunct of θ; if $\phi \vee (t_1 \in \mathit{Tid} \wedge \hat{p}_{t_1} = \mathtt{A} \wedge \check{p}_{t_1} \neq \mathtt{A})$ is valid, we make $\hat{A} \not\subseteq \check{A}$ a conjunct of θ. We call this systematic process *free comprehension-cardinality instantiation*, since such checks instantiate the comprehensions (under cardinalities) by free variables occurring in ϕ.

For our running example, we would like to determine what membership and subset relations hold under

$$\neg\phi \quad = \quad \begin{pmatrix} |\{t \mid t \in \mathit{Tid} \wedge pc_t = \mathtt{A}\}| \le \mathsf{count} \ \wedge\ t_1 \in \mathit{Tid} \ \wedge\ pc_{t_1} = \mathtt{A} \ \wedge \\ \text{other conjuncts} \ \wedge\ |\{t \in \mathit{Tid} \mid (pc[t_1 \mapsto \mathtt{B}])_t = \mathtt{A}\}| > \mathsf{count}-1 \end{pmatrix}.$$

Since this assumption is still not in AUFLIA, we weaken the assumption:

$$\mathit{true} \ \wedge\ t_1 \in \mathit{Tid} \ \wedge\ pc_{t_1} = \mathtt{A} \ \wedge\ \text{other conjuncts} \ \wedge\ \mathit{true}\,.$$

Under this weaker assumption we derive membership and subset relations; for instance we check the validity of the following three formulas:

$$(t_1{\in}\mathit{Tid} \wedge pc_{t_1}{=}\mathtt{A} \wedge \text{other conjuncts}) \;\Rightarrow\; t_1 \in \{t \mid t{\in}\mathit{Tid} \wedge pc_t = \mathtt{A}\}\,,$$
$$(t_1{\in}\mathit{Tid} \wedge pc_{t_1}{=}\mathtt{A} \wedge \text{other conjuncts}) \;\Rightarrow\; t_1 \notin \{t \mid t{\in}\mathit{Tid} \wedge (pc[t_1 \mapsto \mathtt{B}])_t = \mathtt{A}\}\,,$$
$$(t_1{\in}\mathit{Tid} \wedge pc_{t_1}{=}\mathtt{A} \wedge \text{other conjuncts}) \;\Rightarrow\; \forall t\colon \begin{pmatrix} (t{\in}\mathit{Tid} \wedge (pc[t_1{\mapsto}\mathtt{B}])_t = \mathtt{A}) \\ \Rightarrow (t{\in}\mathit{Tid} \wedge pc_t = \mathtt{A}) \end{pmatrix}.$$

These checks succeed, so we obtain $t_1 \in \hat{A}$, $t_1 \notin \check{A}$ and $\check{A} \subseteq \hat{A}$ as additional assumptions. Since the process systematically tries out all possibilities for (non-)membership and (non-)inclusion, the process may also generate some formulas for which subsequent validity checks fail. For instance, since

$$(t_1{\in}\mathit{Tid} \wedge pc_{t_1}{=}\mathtt{A} \wedge \text{other conjuncts}) \;\Rightarrow\; \forall t\colon \begin{pmatrix} (t{\in}\mathit{Tid} \wedge pc_t = \mathtt{A}) \Rightarrow \\ (t{\in}\mathit{Tid} \wedge (pc[t_1 \mapsto \mathtt{B}])_t = \mathtt{A}) \end{pmatrix}$$

cannot be proven valid, $\hat{A} \subseteq \check{A}$ will not be a conjunct of θ.

By construction, we have $(\neg\sigma(\psi)) \Rightarrow \sigma(\theta)$. By Prop. 1, the validity of $\psi \vee \neg\theta$ implies the validity of ϕ. In our running example $\theta = (\check{A} \subseteq \hat{A} \wedge t_1 \in \hat{A} \wedge t_1 \notin \check{A} \wedge \ldots)$ and we check the validity of $\psi \vee \neg\theta$, which is propositionally equivalent to

$$\begin{aligned}&\big(|\hat{A}| \leq \mathsf{count} \wedge t_1{\in}\mathit{Tid} \wedge pc_{t_1}{=}\mathtt{A} \wedge \text{other conjuncts} \wedge \check{A}{\subseteq}\hat{A} \wedge t_1{\in}\hat{A} \wedge t_1{\notin}\check{A}\big)\\ &\Rightarrow\; |\check{A}| \leq \mathsf{count} - 1\,.\end{aligned}$$

Notice that $\psi \vee \neg\theta$ is weaker than ψ and has no top-level cardinalities of comprehensions. Thus, $\psi \vee \neg\theta$ is both valid with a higher chance (than ψ) and nearer to BAPA (than ϕ). This formula is further converted towards BAPA by, for example, modeling elements as singleton sets and substituting atomic formulas with remaining cardinalities of comprehensions (under a binder) by *true* or *false* in a way that is sound for validity. After conversion, we are left with a pure BAPA formula, which is then transformed into the Presburger Arithmetic and fed to an off-the-shelf solver. In our running example the formula $\psi \vee \neg\theta$ could be shown valid by the described tool chain.

The described steps are reminiscent of the theoretical result of Wies et al. [49]. Without going into details, their method, starting with ϕ, universally projects out the non-BAPA variables using different decision procedures, builds a disjunction out of the projections and then checks the disjunction for validity by a BAPA decision procedure. Broadly speaking, one can view our method as a practical adaptation of the theoretical ideas of Wies et al.

The process involving free comprehension-cardinality instantiation leads to success for five out of six proof obligations (full listings are available [32]). Here is the remaining obligation (that comes from checking the invariance of the second conjunct of J with respect to the transition from `A` to `B`) in a simplified form:

$$\begin{pmatrix} |\{t{\in}\mathit{Tid} \mid pc_t{=}\mathtt{A}\}| \leq \mathsf{count} \;\wedge\; (\forall t{\in}\mathit{Tid}\colon (pc_t{=}\mathtt{C} \Rightarrow \mathsf{count}{=}0)) \;\wedge \\ t_1{\in}\mathit{Tid} \;\wedge\; pc_{t_1}{=}\mathtt{A} \;\wedge\; t_2{\in}\mathit{Tid} \;\wedge\; (pc[t_1{\mapsto}\mathtt{B}])_{t_2}{=}\mathtt{C} \;\wedge\; \text{other conjuncts} \end{pmatrix} \Rightarrow\; \mathsf{count}{-}1 = 0\,.$$

A human quickly sees that the antecedent is not satisfiable, so the implication is valid. Its mechanical proof requires an interplay between heuristic quantifier instantiation, the theory of uninterpreted functions and very basic facts about cardinalities:

1. a cardinality is always nonnegative and
2. the cardinality of a set is zero only if the set is empty.

To cope with the last obligation, we translate finite sets of threads into arrays **int**$\to$**int** as follows. Threads are represented by an uninterpreted predicate $\mathsf{IsAThread}$ on integers (the intended meaning is that the predicate is true for integers representing a thread and false for all other integers). Sets of threads are represented by an uninterpreted predicate $\mathsf{IsASetOfThreads}$ on arrays with the assumption $\mathsf{IsASetOfThreads}(a) \Rightarrow (\forall i\colon (a(i)=0 \vee (a(i)=1 \wedge \mathsf{IsAThread}(i))))$. The cardinality is a function on the arrays that satisfies the two aforementioned axioms for arrays a for which $\mathsf{IsASetOfThreads}(a)$ holds. By plainly rewriting the proof obligation into the AUFLIA syntax with the mentioned uninterpreted predicates and assumptions on them we were able to prove the obligation with off-the-shelf SMTLIB solvers.

III. Inductive invariant.
For the first method, which involved translation of the barrier into a program running on sets, we required the following formula

$$I = (A\cup B\cup C\subseteq Tid \wedge A\cap B=\emptyset \wedge A\cap C=\emptyset \wedge B\cap C=\emptyset \wedge \mathsf{count}\geq |A| \wedge (C\neq\emptyset \Rightarrow \mathsf{count}=0))\ .$$

It encodes the following facts:
- A, B, C represent a partition of a set of threads;
- A contains at most count threads;
- If there are threads in C, then $\mathsf{count} = 0$.

The formula lies completely in BAPA. We have proven that it is an inductive invariant manually and, independently, in an automatic verifier, as we will describe below.

For the second method, the inductive invariant is just J.

IV. Encoded program and its invariant.
We will show the input to a verification tool for each of the two methods.

For the first method, the previously described transformations lead to the following input for the Jahob verification system [27], which has an implemented decision procedure for BAPA.

```
int count; // initially the number of participating threads
Element t; // the thread identifier
Set A=Tid, B=∅, C=∅; // modeling the threads at different locations
while(true) //: invariant I = ... (see above)
{
  havoc t∈Thread; // give a nondeterministic value to t.
  if(t∈A) {
    count:=count − 1;
    A:=A \ {t};
    B:=B ∪ {t};
  } else if(t∈B) {
    if(count==0) {
      B:=B \ {t};
      C:=C ∪ {t};
    }
  }
}
```

The command "**havoc** t" chooses an arbitrary thread, the case split chooses one of its two instructions. The code is a sequential one: the threads have been replaced by nondeterminism. This code can be obtained from the original one by a purely syntactic transformation.

The invariant I is attached as a loop invariant to the program above, i.e. on every loop entrance I should hold.

For the second method, the standard encoding of a multithreaded program as a nondeterministic one leads to the following input for Jahob (which also has backends to the SMT solvers Z3 and CVC3).

```
int count; // initially the number of participating threads
Thread t; // the thread identifier
while(true) //: invariant J = ... (see above)
{
  havoc t∈Thread;
  if (t.pc == A) {
    count := count − 1;
    t.pc := B;
  } else if (t.pc == B) {
    if (count == 0) t.pc := C;
  }
}
```

For the second method, we note that Z3 supports more than just SMTLIB, e.g. arrays from an uninterpreted sort Thread into Booleans. If these extensions become standard in SMTLIB, the verification process would get even cleaner by translating sets into arrays Thread → **bool**. Note that the second method reuses a well-understood BAPA logic in all cases where it suffices and introduces cardinality axioms only in a very limited way; to compare, SPASS+T [43] manages whole 12 axioms about counting in collections. We can keep the trust base small by systematically using the free comprehension-cardinality instantiation for BAPA for discharging almost all proof obligations.

V. Using the tool.
Now we describe how we have enhanced and used Jahob for the verification.

We have found and improved several issues in Jahob.

- BAPA allows reasoning about sets, but not about their elements. In BAPA, elements have to be encoded as singleton sets, e.g.: $(\forall t\colon (t \in A \Rightarrow \ldots))$ should be rewritten as $(\forall T\colon ((|T| = 1 \wedge T \subseteq A) \Rightarrow \ldots))$. We have found an error in the conversion and repaired it.
- The decision procedure for BAPA required a single formula in prenex form, i.e. all quantifiers have to precede all other operations, as in $\forall A \exists B\colon A \subseteq B$. The verification process in our case has started producing (internally) formulas not in prenex form, e.g. $(\mathsf{count} \geq |A| \wedge \forall T\colon (|T| = 1 \Rightarrow \ldots))$. We have implemented the conversion to prenex form.
- Checking validity in BAPA is triply exponential in the number of quantifiers, so heuristics are required to overcome time and space limitations of real machines. We have implemented Boolean constant propagation, which reduces the number of quantifiers in certain cases: e.g. $\phi \wedge (\forall T\colon (\ldots \Rightarrow \mathit{true}))$ gets simplified to ϕ and $\exists x\colon (\ldots \wedge \mathit{false})$ gets simplified to *false*. Such optimizations allow approximation of BAPA formulas by sufficiently precise QFBAPA (quantifier-free BAPA) formulas. Validity in QFBAPA has a much lower NP complexity.

After these improvements, Jahob was able to verify the program by the first method in 724 s on an Intel® Core™ i5-520M CPU clocked at 2.4 GHz with 8 GB RAM clocked at 1066 MHz.

We have completely implemented the free comprehension-cardinality instantiation in Jahob. We are in the process of implementing the aforementioned automatic syntactic translation of proof obligations into SMTLIB 2.0 with axioms for sets and cardinalities. There are several engineering difficulties in this translation, the main ones arise from the combination of higher-order logic with the potential presence of polymorphic operators like unions and intersections and with a potentially incomplete typing information (the type information sometimes has to be dropped for the sake of saving memory). For example, the subformulas of the formula $|\mathbf{let}\, x = \cup\ \mathbf{in}\ (x\ y1\ y2\ y3)|$ cannot be monomorphically typed, hence its conversion to SMTLIB 2.0 is hard. At the same time, $|\mathbf{let}\, x = \cup\ \mathbf{in}\ (x\ y1\ y2\ (y3 :: (\mathsf{Thread}\ \mathbf{set})))|$ can be translated as $\mathit{card\_of\_threadSet}(\mathbf{let}\ x = \mathit{union\_of\_three\_threadSets}\ \mathbf{in}\ (x\ y1\ y2\ (y3 :: (\mathbf{array\ int\ int}))))$, where *union_of_three_threadSets* are *card_of_threadSet* are fresh variables that are unique for the whole proof obligation and that map three integer arrays to an integer array and an integer array to an integer. These variables model the union of three sets of threads and the cardinality of a set of threads, and axioms can furnish such variables with appropriately sound and sufficiently complete semantics. We have found an acceptable engineering solution and are proceeding further towards generating SMTLIB 2.0 scripts.

All the proof obligations from the second method except the last one are handled altogether in about 87 s on an Intel® Core™ i5-520M CPU clocked at 2.4 GHz with 8 GB RAM clocked at 1066 MHz. The manual translation of the last one is handled by Z3 in less than a second.

VI. Beyond the frontier.
The following challenges remain for each of the two methods.

In the first method, a slight syntactic change of the invariant allows us to see the property to be verified more directly: we state that if some thread t is as at location C, then count is zero. The following property could not be verified: $\tilde{I} =$

$$A \cup B \cup C \subseteq \mathit{Tid} \wedge A \cap B = \emptyset \wedge A \cap C = \emptyset \wedge B \cap C = \emptyset \wedge \mathsf{count} \geq |A| \wedge (\forall t\colon\ (t \in C \Rightarrow \mathsf{count} = 0)).$$

The additional quantifier in $\tilde{I}$ leads to an out-of-space error in the tool chain. We recognize a better decision procedure as the next improvement step [48].

The challenge for the second proof method is the full automation of the translation of cardinalities of comprehensions into SMTLIB.

4.2 Central barrier coded in C

We continue with an actual implementation of the central barrier in multithreaded C. This implementation is taken from `sequent/barriers/central.c` [37].

I. Code.
The considered executable implementation below can be called more than once in a run; in the following, an *episode* is a single execution of the barrier synchronization code.

As in Section 4.1, the implementation is organized around the shared variable count that counts the number of threads that have not yet synchronized. Additionally, the implementation contains a single shared Boolean variable sense and thread-private Boolean variables $\mathsf{local\_sense}$ that track whether the number of executed episodes in an execution is even or odd. When starting an episode, a thread first stores the parity of the new episode into a thread-private variable, then decrements count and checks whether the decrement was from 1 to 0. If it was so, the thread reinitializes count for the next episode, updates the shared parity of the number of episodes (thus waking up the other threads) and finishes the current episode. If the decrement was from a value greater than 1, the thread waits until the shared parity is modified and only then finishes execution of the barrier code. The thread detects when sense gets toggled by comparing it with a value of the private $\mathsf{local\_sense}$.

```
shared int count = num_nodes; // num_nodes is the number of threads.
shared bool sense = false; // the shared parity of the number of episodes.
local int local_count; // temporary variable.
local bool local_sense = false; // A wake-up is signaled by equaling the variables local_sense and sense.
void barrier() {
  local_sense^= true; // setting the parity of the new episode locally.
  // The number of threads that have not yet arrived at the barrier decreases:
  local_count = fetch_and_decrement(&count);
  if(local_count>1) goto busy_wait; // if more threads will arrive, start waiting.
  // Otherwise the current thread was the last to arrive.
  count = num_nodes; // reinitialize count.
  sense = local_sense; // signal wake-up.
  return; // and exit.
  busy_wait: while(sense!=local_sense); // wait for the wake-up.
}
```

This code is a little different from the one found in `central.c`, which is a mixture of assembler and C specific to a particular compiler. We have slightly changed the code by rewriting assembler into modern C and using fetch_and_decrement, which can be compiled into a single machine instruction for some instruction sets like those of IA64. (Compare-and-swap would work as well, except requiring a loop.)

II. Required reasoning.
The barrier property we verify is: when some thread finishes executing barrier(), all the other threads have at least started executing their barrier() calls. We will show what kind of reasoning is required for a proof in a verifier for C code.

VCC (Verifying C Compiler) is a verifier that supports ANSI C syntax with multi-threaded semantics. It has been used to prove properties of Microsoft hypervisor [28], so we consider VCC mature enough for our purpose. To verify that a certain invariant of a program holds, VCC internally generates Boogie code, which then gets verified by calls to a first-order automatic theorem prover Z3. However, the default tool chain VCC-Boogie-Z3 cannot directly handle the cardinality constraints needed for the straightforward proof similar to one in Section 4.1.

An aside should be made. Quantifier-free cardinality constraints can be handled in a plugin for Z3 [48]. However, this plugin is (at least as of Spring 2012) not publicly accessible. Even if it were, both Boogie and VCC lack syntactic constructs for sets and their cardinalities, let alone their semantic translation.

Though it is possible to represent a set of integers as a map `int`$\to$`bool`, and it is possible to express the cardinality of this set, namely as a recursive function computing the size of the support of this map, it is hard to reason about this recursive function in an automatic way. (Before summer 2012 it was completely impossible. In summer 2012 VCC started supporting termination proofs of recursive functions. However, the user would still have to define the notion of cardinality and reason about it.) One should note that a non-automatic way existed in the past, namely a non-default translation from Boogie into the interactive theorem prover Isabelle/HOL producing 600 Kb of Isabelle/HOL code which has to be verified interactively. Our solution uses neither recursive functions nor Isabelle/HOL. Instead, we use an approach (communicated to us by Ernie Cohen, the core ideas are due to Georg Cantor) that turns the recursion in the definition of cardinality into the induction over the execution length, since this induction is implicitly built into the proof rule for invariance properties. When following this approach, we will augment the program with auxiliary variables in such a way that the augmented program together with an augmented property lies inside the fragment that the VCC-Boogie-Z3 tool chain can handle automatically. This tool chain works well for inductive invariants of the form $\forall \vec{x}\colon\ \phi(\vec{x})$ where ϕ is a quantifier-free formula that directly lies in or can be easily translated into the logic QF_AUFLIA (the quantifier-free theory of arrays, uninterpreted functions and linear integer arithmetic) and $\vec{x}$ is a finite vector of variables. We will write an inductive invariant in such a form.

The auxiliary code does not influence the original control flow or the values of the original variables; removing the auxiliary code results in the original barrier code. For a normal C compiler, this auxiliary code is `#define`d as an empty string, so the auxiliary code has no influence on compilation; for VCC, the auxiliary code has a proof-theoretic meaning.

III. Inductive invariant.
In this section, we will overcome the main difficulty: dealing with the cardinality of a set of threads. For that, we will start with an informal notice, then describe our idea, illustrate it on an example and finally show the high-level structure of the inductive invariant.

First notice that the only relevant set seen in the invariant of the previous Section 4.1, namely the set A of all threads that have not yet executed the decrement, shrinks by one element or stays the same during a transition of every thread t. Thus, the verification conditions (which speak about either the initial state or the pre- and post-state of a single transition) have to compare cardinalities of syntactically different sets A and $\hat{A}$ where A and $\hat{A}$ semantically satisfy either $A = \hat{A}$ or $A = \hat{A} \setminus \{t\}$. The cardinality of these sets either remains unchanged or drops by one at any transition of an execution; this fact, informally, enables reusing induction built into the definition of an execution.

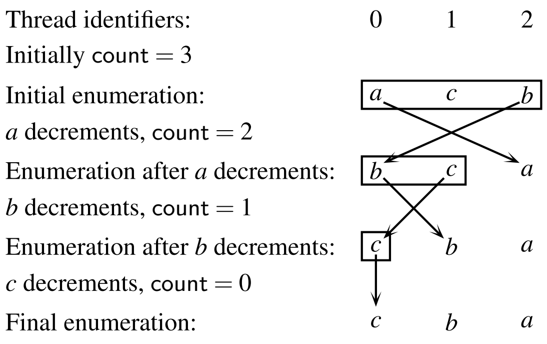
Now we introduce the *Cantor-Cohen approach*. Briefly, the approach describes how a bijection enables us to use the induction that is built into the definition of an execution—and thus also built into inductive-invariant–based verification—to reason about the cardinalities (of thread sets). (To the best of our knowledge, the idea of capturing set cardinality using a bijection is due to Georg Cantor. Though this idea is necessary for infinite sets, we profit from it also for finite sets. The application of the idea to multithreaded programs is due to Ernie Cohen.)

Remember that a cardinality of a finite set is just a natural number (here, including zero), and natural numbers are used to enumerate objects, for example, threads, by attaching unique identifiers to them. We will enumerate threads in such a way that the proof obligations will just speak about thread identifiers instead of cardinalities of A and $\hat{A}$.

Here induction over execution length enters the game. Given an execution, we enumerate the threads according to the order in which they pass the most important instruction fetch_and_decrement. The enumeration will use consecutive natural numbers in descending order: the last thread receives identifier 0, the last but one receives identifier 1, and so on, while the earliest thread that passes the barrier gets identifier $\mathsf{num\_nodes} - 1$. Remark the following property: the identifier of a thread that has not yet decremented is below count. However, the enumeration depends on the execution, which is unknown beforehand; thus we construct the enumeration during execution. The construction is easy: right after the decrement we make sure that the decrementing thread has identifier count by exchanging its current identifier (which is less than or equal to count) with a thread whose current identifier is count. As we will see, the exchange code can be made independent of the execution.

A worked out example using the Cantor-Cohen approach. Assume that three threads a, b, c, are arbitrarily bound to the first three naturals, say, as in bijection $s_0 = [0 \mapsto a, 2 \mapsto b, 1 \mapsto c]$ and decrement in alphabetical order: first a, then b, then c. Note that initially all thread identifiers are bound to naturals that stay below the current value of count, which is 3. Right after thread a decrements, the identifier of a is swapped with that of $s_0(\mathsf{count}) = s_0(2)$

$= b$, which results in bijection $s_1 = [2 \mapsto a, 0 \mapsto b, 1 \mapsto c]$. Notice that in s_1 the identifiers of all threads that have not yet decremented (namely, b, c) are below the current value of count, which is 2. Right after thread b decrements, the identifier of b is swapped with that of $s_1(\mathsf{count}) = s_1(1) = c$, which results in bijection $s_2 = [2 \mapsto a, 1 \mapsto b, 0 \mapsto c]$. Observe that in s_2 the identifier of the single thread that has not yet decremented (namely, c) is below the current value of count, which is 1. At last, thread c decrements, which already has the right identifier 0, so swapping does not change the bijection, and all the threads have decremented. Adapting the enumeration to the schedule can be visualized as follows:


Thread identifiers: 0 1 2
Initially count = 3
Initial enumeration: a c b
a decrements, count = 2
Enumeration after a decrements: b c a
b decrements, count = 1
Enumeration after b decrements: c b a
c decrements, count = 0
Final enumeration: c b a

We maintain the bijection as two auxiliary variables *states* and *states_back* that represent mutually inverse maps, where *states* maps a thread identifier (a number from $[0, \mathsf{num\_nodes})$) to the pointer to the private state of the thread. (In the sequel, the auxiliary code and the comments will be typeset in *italics*). This map allows obtaining the control flow location of a thread from a thread identifier. We maintain the invariant that *states* is one-to-one ($\forall x \neq y\colon\ states[x] \neq states[y]$), and that *states_back* is its right inverse. A direct claim that *states* is onto would require an existential quantifier. Thus it would be problematic for Z3 to prove the validity of such a formula. If one would like to assert surjectivity instead of the existence of the right inverse, one would additionally need either to correctly instantiate the axiom of choice or to inductively construct the right inverse.

We reduce the number of control flow locations that have to be reasoned about by maintaining abstract control flow locations instead of the real ones:

APL	before writing local_sense
BPL	just before fetch_and_decrement
CPL	just before checking local_count > 1 and at busy_wait,
WakeUpPL	during reinit of sense and count and after busy_wait,

ordered as $APL < BPL < CPL < WakeUpPL$.

The inductive invariant is a conjunction of three facts. The first two conjuncts say that *states* is one-to-one and *states_back* is its right inverse. The third part is a disjunction of two formulas:

- The first disjunct elucidates the synchronization phase. It is a conjunction of several facts about all thread identifiers t with $states[t] \neq 0$ (i.e. about all the threads). The most important conjuncts are the following:
  - $states[t] \to pc \leq CPL$, i.e. t is in the synchronization phase,
  - $states[t] \to pc \leq BPL \Rightarrow t < \mathsf{count}$, i.e. if a thread has not yet decremented, then its identifier is below count.
- The second disjunct establishes that each thread is waiting or is waking up: $\forall t \in Tid\colon$ $states[t] \neq 0 \Rightarrow CPL \leq states[t] \to pc$, where Tid is the set of nonnegative integers (modeling thread identifiers).

IV. Encoded program and its invariant.
The input to VCC will consist of the declaration of the state including the auxiliary variables and of the barrier function together with assignments to the auxiliary variables; all parts of the input will be annotated with formulas. We will show and explain the input in chunks.

The difficulty is that VCC allows syntactically attaching invariants to:

- control flow locations. An annotation of a control flow location describes the state of a thread at that given control flow location.
- C structures. An annotation of a C structure describes the values of its instances between atomic transitions.

To distribute our inductive invariant to these places, we create appropriate structures.

The structure ThreadState below holds the thread-private state of a thread. It contains auxiliary variables *pc* and *local_count*, allowing the global invariant to speak about the abstract control flow location of the thread and the value of the actual procedure-local variable local_count.

```
typedef enum PL {APL,BPL,CPL,WakeUpPL} PL; // type of abstract control flow locations.
typedef unsigned Tid; // thread identifiers.
// To reason about private state in a structure invariant, we have to pack this private state into a structure.
typedef struct ThreadState {
  unsigned local_sense; // modeling a Boolean in ANSI C.
  PL pc; // abstract control flow location.
  unsigned local_count; // copy of the corresponding private variable.
} *PThreadState;
```

Booleans local_sense and sense (below) get a C integral type.

The structure Instance below contains the variables count, sense, a constant denoting the number of threads and the maps *states* and *states_back* that associate thread identifiers with pointers to private states. In the following code, a variable specified as **volatile** or *volatile* can potentially be changed by more than one thread.

```
typedef struct Instance {
  const unsigned num_nodes; // the number of threads, initialized elsewhere.
  volatile unsigned count; // threads that have not yet started.
  volatile unsigned sense; // toggles from one episode to the next one.
  volatile PThreadState states[Tid]; // maps thread identifiers to pointers to threads' private states.
  volatile Tid states_back[PThreadState]; // maps pointers to threads' private states to thread identifiers.
};
```

The fetch_and_decrement instruction is not a part of ANSI C, being unknown to VCC. We make VCC aware of the semantics of fetch_and_decrement:

```
atomic_inline // making VCC aware that the function is atomic.
unsigned fetch_and_decrement(volatile unsigned* address) {
  return (*address)−−; /* can be implemented atomically on Itanium x64 as
                          "fetchadd4.rel retval=address,−1" or by a compare-and-swap in a loop. */
}
```

Now we will look at the instrumentation of the barrier function. In the code below, the brackets $\langle \ldots \rangle$ denote atomic transitions. Each such atomic transition contains at most one physical instructions on shared variables and arbitrarily many instructions on auxiliary variables.

The barrier procedure below will require some explanation. The procedure has three procedure-local thread-private variables. The variable local_count serves two purposes: first, it stores the fetched value thread-privately, second, it is used as a temporary location to reinitialize count of the Instance structure. The flag still_wait controls breaking or continuation of the waiting loop, new_local_sense serves toggling local_sense of a ThreadState structure.

VCC does not allow reasoning in structure invariants about procedure-local variables or the control flow. Thus we keep track of the important procedure-local variable local_count and the abstract control flow in auxiliary fields *local_count* and *pc* of the structure ThreadState. VCC does allow reasoning about the fields of ThreadState (after some additional VCC-specific annotations which we will skip in this presentation) in the invariant of the Instance structure.

The barrier code starts by reading local_sense of its ThreadState instance *pt and storing the toggled value into the local variable new_local_sense. Then, the code updates local_sense from new_local_sense. The annotation tracks this fact by changing the abstract control flow location from *APL* to *BPL*.

After that, the code fetches the current value of count into the procedure-local variable local_count and decrements count. The maps *states* and *states_back* are updated according to the Cantor-Cohen approach above: the update ensures that all the threads that have not yet executed fetch_and_decrement have identifier below count, formally

$$\forall t \in Tid\colon\ \left((states[t] \neq 0\ \wedge\ states[t] {\to} pc \leq BPL)\ \Rightarrow\ t < \mathsf{count} \right).$$

Suppose that this condition holds before fetch_and_decrement. The running thread establishes this condition afterwards by

- exchanging the identifier of the running thread (this identifier will be stored in the variable *myId* in the code) with count (which is the identifier of the thread whose private state will be stored at *targetOfLast* in the code);
- updating the abstract control flow location to *CPL*.

If the exchange would not happen, the thread with private state **targetOfLast* could start violating the above condition. To see the counterexample, imagine that the thread with private state **targetOfLast* is different from the running thread and has not yet decremented. The just mentioned operations (exchanging the thread identifier and updating the abstract control flow location) reestablish the condition.

After that, the fetched value is tested. If it indicates that more threads are still going to decrement, the current thread has to wait for the last thread in a loop at location busy_wait. Otherwise the fetched value is 1, so no more threads will arrive, and the current thread can start waking up.

The wake-up proceeds by first updating the abstract control flow location to *WakeUpPL*. Then the wake-up copies num_nodes to count (which cannot happen atomically, since it requires dereferencing the structure Instance) by using a procedure-local temporary variable, for which we utilize local_count. Finally, the wake-up updates sense, which cares for the shared parity of the number of episodes, thereby waking up the other threads, and goes to the final control flow location Last.

If a thread has to wait in the loop at busy_wait, it constantly checks whether sense has finally got the updated parity, which is now stored in new_local_sense. If so, the code updates the abstract control flow location to *WakeUpPL* and sets still_wait to false, which results in jumping out of the loop. After quitting the loop, the thread proceeds to the final control flow location Last.

At the final control flow location, the barrier property is asserted, i.e. for all valid thread identifiers t (i.e. that have nonzero $states[t]$), the abstract control flow location is no longer *APL*.

```
void barrier(struct Instance *s, PThreadState pt)
requires pt→pc == APL // assuming that the abstract location variable has the value APL initially.
{
  unsigned local_count; // used for fetch-and-decrement.
  unsigned char still_wait=1; // used for the condition of the loop.
  unsigned new_local_sense; // used for toggling local_sense.
  /* Toggling local_sense in pt cannot be done atomically, since one has to
   read pt too. A thread reads, toggles privately and writes in two steps. */
  ⟨ new_local_sense = (pt→local_sense)^1; ⟩ // read local_sense and toggle it.
  ⟨ (pt→local_sense)=new_local_sense; pt→pc=BPL ⟩ // write the altered local_sense back into *pt.
  ⟨ myId = s→states_back[pt]; // get the identifier of the thread.
    // Fetch the count and decrement it afterwards—in one atomic step:
    local_count = fetch_and_decrement(&(s→count));
    // The thread's identifier is ≤ count, swap it with count as follows.
    // Determine the private state of the thread with identifier count:
    PThreadState targetOfLast = s→states[s→count];
    // Update the maps states and states_back:
    s→states = swap(s→states, myId, s→count);
    s→states_back = swap(s→states_back, targetOfLast, pt);
    // Store the fetched value and update the abstract program counter:
    pt→pc = CPL; pt→local_count = local_count; ⟩
  // Test the fetched value:
  if(local_count>1) { // If more threads decrement after this one:
    goto busy_wait; // proceed to a waiting location.
  } else {⟨ pt→pc = WakeUpPL; ⟩} // otherwise reinit & wake up.
  // Reinitialization is reading the number of threads and writing count.
  // This is a non-atomic operation on shared state, use local_count as temporary thread-private storage.
  local_count = s→num_nodes; // reading s→num_nodes.
  s→count = local_count; // writing s→count.
  s→sense = new_local_sense; // writing the toggled sense.
  goto Last; // reinitialization finished.
busy_wait: // sleeping location, a thread waits for the wake-up here.
  while(still_wait) // Either waiting or waking up:
  invariant still_wait || (pt→pc == WakeUpPL)
  { // Wait until the last arriving thread toggles sense:
    ⟨ if(s→sense == new_local_sense) {
          still_wait = 0; // if sense got toggled, jump out of the loop,
          pt→pc=WakeUpPL; // waking up.
       }
    ⟩
  }
Last: // The property we want to check:
  assert (∀t ∈ Tid :  s→states[t]≠0 ⇒ s→states[t]→pc≠APL)
}
```

The inductive invariant that was loosely introduced on page 15 can now be stated precisely (in this formula we use the C syntax, in which "==" means equality testing):

$$I = \left(\begin{array}{l}
(\forall\, i,j \in \mathit{Tid}\colon\ (\mathit{states}[i] == \mathit{states}[j] \neq 0) \ \Rightarrow\ i == j) \ \wedge \\
(\forall\, p \in \mathsf{PThreadState}\colon\ \mathit{states}[\mathit{states\_back}[p]] == p) \ \wedge \\
\left(\left(\left(\forall t \in \mathit{Tid}\colon \left(\mathit{states}[t] \neq 0 \Rightarrow \left(\begin{array}{l}
\mathit{states}[t] \to \mathit{pc} \leq \mathit{CPL} \\
\wedge \\
(\mathit{states}[t] \to \mathit{pc} \geq \mathit{BPL} \ \Leftrightarrow\ \mathit{states}[t] \to \mathsf{local\_sense} \geq 1) \\
\wedge \\
(\mathit{states}[t] \to \mathit{pc} < \mathit{CPL} \vee \mathit{states}[t] \to \mathit{local\_count} > \mathsf{count}) \\
\wedge \\
(\mathit{states}[t] \to \mathit{pc} \leq \mathit{BPL} \ \Rightarrow\ t < \mathsf{count}) \\
\wedge \\
(\forall \hat{t} \in [0, \mathsf{count})\colon \mathit{states}[\hat{t}] \neq 0)
\end{array}\right)\right)\right) \right.\right. \\
\quad \left.\wedge\ \mathsf{sense} == 0 \right) \\
\quad \vee \\
\quad \left.(\forall\, t \in \mathit{Tid}\colon\ \mathit{states}[t] \neq 0 \Rightarrow \mathit{CPL} \leq (\mathit{states}[t] \to \mathit{pc}))\right)
\end{array}\right).$$

The invariant textually annotates the Instance structure.

The big last conjunct of the invariant implies that when some thread finishes, i.e. arrives at the location *WakeUpPL*, all threads are in the wake-up phase; in particular, no thread is at the initial control flow location *APL*.

V. Using the tool.
VCC verifies the inductive invariant in 2.92 s on an Intel® Core™ i5-520M CPU clocked at 2.4 GHz with 8 GB RAM clocked at 1066 MHz and 3.427 s on an Intel® Core™2 Duo L7500 CPU clocked at 1.6 GHz with 1.5 GB RAM clocked at 533 MHz. Due to the constant development of VCC, it is much more user-friendly than any other tool we used for barrier verification.

VI. Beyond the frontier.
The encoding was constructed around the auxiliary variable *states*. Its purpose is, from a high-level view, to replace recursion in the cardinality definition by induction over the trace length, supported by more user-defined auxiliary variables. Simplifying the annotation would require adding a direct decision procedure for reasoning about set cardinalities into the VCC-Boogie-Z3 tool chain. We are ready to target this challenging goal in future research.

4.3 Jahob+BAPA+AUFLIA versus VCC+Boogie+AUFLIA

In this subsection we evaluate our experiences in the approaches (two via Jahob and one via VCC) to verify similar central barriers.

The first method of handling the simplest counter barrier in Jahob involved syntactically rewriting a multithreaded program into a program operating on sets. The verification conditions were fully handled by a decision procedure for BAPA. This approach does not require writing any additional axioms—the logic itself is powerful enough to handle sufficiently complicated cardinality constraints. In this respect this approach is the cleanest one: the trust base for verification is small because the user just has to believe in the correctness of the implementation of the decision procedure with respect to the definition of the logic. The price paid is that the user has to decide (in her own mind) on one bit of information prior to verification: namely, whether the multithreaded program should be converted into one operating on sets. On the positive side, less program abstraction occurs in comparison to the counter abstraction method.

The second method of handling the simplest counter barrier in Jahob involved just viewing the program as a nondeterministic one—without any other conversions at all. All but one verification conditions were adapted to BAPA by a new systematic method: the free comprehension-cardinality instantiation (an AUFLIA solver restored the information about subset inclusion and membership between free variables of the corresponding types). Thus weakened conditions were discharged by the BAPA decision procedure. The one remaining verification condition could be handled by an AUFLIA solver directly under a tiny set of axioms. It is a pleasant feature for a user to avoid deciding on whether a conversion of a multithreaded program to sets should happen or not prior to verification. The price paid is that the trust base for the verification is a bit larger: in addition to believing in the correctness of the implementation, the user also has to believe that the two added axioms are sound for validity. However, the set of added axioms is tiny—it fits into a couple of lines. (We have indeed discovered errors in axioms written by experts in various contexts. A reader is encouraged to take a look at the bugs, say, in the huge prelude of VCC that are mentioned in the VCC [11] bug tracker!) Again, there is no loss of precision prior to the stage where the semantics gets visible in the verification conditions.

The VCC way of handling a more complicated central barrier required getting rid of higher-order cardinality operator via Cantor-Cohen approach. This approach can be generalized: it has nothing special to a particular barrier implementation, just to the fact that we need to express “in an execution, the number of threads at a particular set of locations is decreasing” in AUFLIA. Instead of reasoning about inclusion of sets, about membership of elements in sets and about cardinalities of sets, the Boogie-Z3 back-end had to reason, in its hardest part, about a bijection between pointers to threads’ local states and thread identifiers, which is expressible in AUFLIA. Due to a marvelous work of the VCC team, their verifier is by far the easiest to use. The paid price is the mental effort the user has to invest to apply the approach correctly to come up with a suitable invariant and auxiliary variables. On the positive side, the tool chain accomplishes verification in astonishingly small time, making the process very suitable for on-line debugging. Also, the VCC approach is able to verify an implementation in C rather than just a pseudocode algorithm.

5 Tree-based barriers

The previously described shared-counter barriers are simple, but not the optimal ones. Namely, a decrement of the single shared counter has to be visible to all threads to ensure correctness, so the accesses to the counter should be executed sequentially. Thus, in the best case, all the barriers terminate in at least linear time. In the worst case, contention on the single location can lead to super-linear running time.

There are more complicated algorithms that have better, logarithmic best-case running time. The so-called tree-based barriers employ a shared tree for communication. As we will see, each memory location of the tree is shared only between a constant number of threads, and the barriers have logarithmic best-case execution time. We will verify the most renowned tree-based barriers, namely the static and the combining tree barriers, in Jahob.

5.1 Static tree barrier

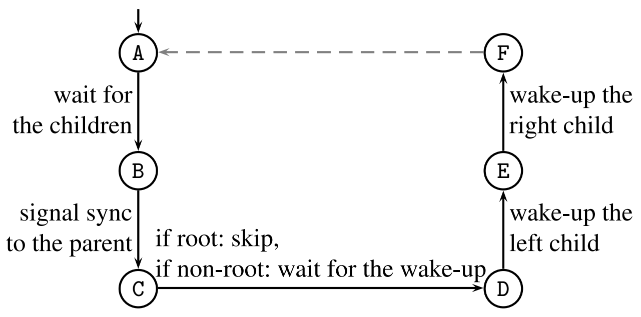
This section handles the verification in Jahob of the static tree barrier, described by Herlihy and Shavit [20].

I. Algorithm.
In the static tree barrier threads operate on a single tree, in which each thread is statically associated with a distinct tree node, e.g.:

$$\begin{tikzcd}
\text{Thread 0} \arrow[dr] & & \text{Thread 2} \arrow[dl] & & \text{Thread 4} \arrow[d] \\
& \text{Thread 1} \arrow[dr] & & & \text{Thread 5} \arrow[dll] \\
& & \text{Thread 3} & &
\end{tikzcd}$$

We assume that threads use one shared tree in which each node has at most two children. Each thread corresponds to a distinct tree node. The mapping of threads to nodes remains constant, this is why the barrier is called static. The whole computation proceeds in two phases. A thread (say, number 1) starts in the synchronization phase by waiting until its children (0 and 2) have synchronized, then telling the parent (3) that the whole sub-tree (0,1 and 2) rooted at the thread has synchronized and then waiting for the wake-up command from the parent (3). In the wake-up phase, once the parent (3) wakes up the thread (1), the thread wakes up its children (0 and 2) and proceeds with the instructions following the barrier call. The root (3) does not wait for a wake-up at the end of its synchronization phase; the static barrier algorithm guarantees that the root starts waking up only if all the threads have started computation.

A thread's computation can be schematized as follows:



The gray dashed line is worth a minor comment. We present the static tree barrier as a one-time barrier, i.e. without reinitialization, and we will analyze it as a one-time barrier. However, one could easily turn it into a reusable barrier just by merging the initial and the final control flow locations. This change is indicated by the gray dashed line.

The code of the static tree barrier is given below.

```
// Each thread points to a distinct node:
local node *n; // The fields parent, left and right span the tree;
// the field sense, initialized to false, is used for signaling.
// Each thread executes the following function:
void barrier() {
A: // Wait for synchronization of descendants:
   if(n has a left child) while(!(n→left→sense));
   if(n has a right child) while(!(n→right→sense));
B: if(n has a parent) n→sense = true; // signal synchronization to the parent.
C: while(n→sense); // sleep, waiting for a signal from the parent.
D: // Signal wake-up to the left and right children:
   if(n has a left child) n→left→sense = false;
E: if(n has a right child) n→right→sense = false;
F: }
```

For the purpose of presentation we consider each transition between consecutive locations as atomic (otherwise we would just get a larger inductive invariant). The meaning of sense is two-fold:

- in the synchronization phase, $n \to \mathsf{sense} = \mathit{true}$ means "n and all its descendants" have entered the barrier;
- and in the wake-up phase, $n \to \mathsf{sense} = \mathit{false}$ means "n and all its ancestors" are woken up.

Notice that the thread at the root does not sleep at location C: the sense field of the root is initially *false* and the transition B-C does not touch it, since the root has no parent. Thus, at location C, the root still has the cleared sense flag and does not have to wait; the root immediately starts the wake-up phase for its both sub-trees.
As before, we consider the multithreaded program in which each threads starts at location A, executes the body of the barrier function above and does not proceed beyond location F.

II. Required reasoning.
The barrier property we verify is: if some thread arrives at location F, all threads are no longer at location A. To prove this property we would like to reason about trees and unboundedly many threads, which would require an undecidable logic. However, we may use the fact each thread is associated with a unique tree node and speak just about trees. We make the program counter of a thread an explicit field pc of the node corresponding to the thread, thus relieving the invariant from the obligation to speak about threads. The logic that allows us to express the descendant relation in the trees and reason about the node contents is WS2S, the second-order weak monadic logic of two successors. Let T be the set of non-null addresses of tree nodes (from now on, we will write 0 for the null pointer).

III. Inductive invariant.
First we show the most important subformula of the inductive invariant and then the whole inductive invariant.

In the synchronization phase every node n enjoys the following crucial property. A descendant m of n has arrived at location C if one of the two conditions hold: either (a) m is different from n and n has finished waiting for its children (i.e. being at location B or later) or (b) m coincides with n and n has a raised the sense flag. In (syntactically sugared) WS2S, this property is

$$J \quad = \quad \left(\forall m \in T : \begin{pmatrix} (m,n) \in \mathsf{parent}^* \ \wedge \\ ((n \to \mathsf{pc} \geq \mathtt{B} \wedge n \neq m) \vee n \to \mathsf{sense}) \end{pmatrix} \Rightarrow m \to \mathsf{pc} \geq \mathtt{C}\right) ,$$

where parent^* is the transitive-reflexive closure of the one-step parent relation, i.e. it contains all (m,n) such that m is a descendant of n or n itself. Another important property of the synchronization phase is that all threads are at or before location `C`.

The wake-up phase starts when all threads get behind location `C`, i.e. $(\forall n \in T: n \rightarrow \mathsf{pc} \geq \mathtt{C})$. The disjunction of the formulas describing the synchronization and the wake-up phases is the inductive invariant I. To fully specify I, we need to know that during the synchronization phase, a non-root thread that is at location `C` has a raised sense flag: $((n \rightarrow \mathsf{pc} = \mathtt{C} \wedge n \rightarrow \mathsf{parent} \neq 0) \Rightarrow (n \rightarrow \mathsf{sense}))$. Then

$$I \quad = \quad \begin{pmatrix} (\forall n \in T \colon (J \ \wedge \ n \rightarrow \mathsf{pc} \leq \mathtt{C} \ \wedge \ ((n \rightarrow \mathsf{pc} = \mathtt{C} \wedge n \rightarrow \mathsf{parent} \neq 0) \Rightarrow (n \rightarrow \mathsf{sense})))) \\ \vee \ (\forall n \in T \colon n \rightarrow \mathsf{pc} \geq \mathtt{C}) \end{pmatrix}.$$

IV. Encoded program and its invariant.

Jahob is one of the best available tools to reason about trees in WS2S, so we have chosen Jahob to verify the static tree barrier.

Jahob internally calls the decision procedure MONA to solve the underlying WS2S queries. Jahob cannot deal with concurrency natively, but accepts sequential nondeterministic programs which can model concurrent programs. Our nondeterministic program assumes that I holds, chooses an arbitrary tree node n, assumes that $n \rightarrow \mathsf{pc} \in \{\mathtt{A}, \ldots, \mathtt{E}\}$, constructs the next state after the single thread transition and ensures that I holds after the transition. Since the WS2S logic does not admit integer variables, we translate bounded integer variables such as pc to a fixed number of bit variables. The values `A`...`F` are represented bitwise in fields pc_bit_2, pc_bit_1, pc_bit_0.

The transformation of the code into Jahob input is a purely syntactic step. We provide the transformed code in Java syntax below.

Each node (**class** Node) contains the following fields:

– the Boolean flag sense, described above;
– bits pc_bit_2, pc_bit_1, pc_bit_0, which encode the control flow location of the thread at the node as follows: `A`=000, `B`=001, ..., `F`=101;
– pointers parent, left, right to the parent, left child or right child of the node. The parent field of the root has value 0. If the left (resp. right) child of a node is missing, the corresponding field has value 0.

In the code below, the outer loop **while**(**true**) makes the scheduler run forever. The instruction **havoc** k∈T chooses a node (= thread) to execute the next instruction:

```
Node k;
while(true) /*: invariant  ... I above ... */ {
  havoc k∈T; // choose a thread (= node) to execute the next instruction.
  ... // read the control flow location of the thread and execute the command starting at this location.
}
```

Branching statements (like **if**(k.pc_bit_2) { **if**(!(k.pc_bit_1)) { **if**(!(k.pc_bit_0)) ... }}) are used to read the control flow location of the chosen thread bitwise. The innermost statements execute the actual instruction of the nondeterministic program, changing the control flow location if necessary. For instance, consider the instruction

```
if(n has a right child) n→right→sense = false;
```

at location `E` with successor location `F`. The location `E` is represented by the bitstring 100, the location `F` by the bitstring 101. Thus, the translation of this instruction is

```
if(k.right) k.right.sense = false;
k.pc_bit_0 = true;
```

Below we see the full body of the above loop.

```
if(k.pc_bit_2) {
  if(!(k.pc_bit_1)) {
    if(!(k.pc_bit_0)) { // encoding transition E → F
      if(k.right) k.right.sense = false;
      k.pc_bit_0 = true; } }
} else { // ¬(k.pc_bit_2)
  if(k.pc_bit_1) {
    if(k.pc_bit_0) { // encoding transition D → E
      if(k.left) k.left.sense = false;
      k.pc_bit_2=true; k.pc_bit_1=(k.pc_bit_0)=false; // goto E
    } else { // ¬(k.pc_bit_0), encoding transition C → D
      if(!(k.sense)) k.pc_bit_0 = true; }
  } else { // ¬(k.pc_bit_1)
    if(k.pc_bit_0) { // encoding transition B → C
      if(k.parent) k.sense = true;
      k.pc_bit_1 = true; k.pc_bit_0 = false;
    } else { // ¬(k.pc_bit_0), encoding transition A → B
      if(k.left) {
        if(k.right) {
          if((k.left.sense)&&(k.right.sense)) k.pc_bit_0=true;
        } else { if(k.left.sense) k.pc_bit_0 = true; }
      } else {
        if(k.right) {
          if(k.right.sense) k.pc_bit_0 = true; } } } } }
```

The invariant textually annotates the head of the loop.

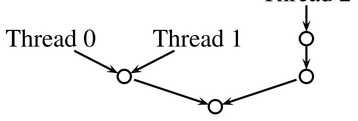
V. Using the tool.
Using the tool is straightforward. Its running time is 15 s on an Intel® Core™2 Quad Q9550 CPU clocked at 2.83 GHz with 8 GB RAM clocked at 800 MHz. Notice that the static tree algorithm was verified for all trees, i.e. for any number of participating threads.

VI. Beyond the frontier.
One nontrivial step was encoding the local state of the threads into the nodes of a single barrier tree. However, if we need to reason about more than one barrier tree, then we would need to make threads first-class citizens. Reasoning about such programs would be a challenge: extending WS2S with an unbounded number of threads makes the logic undecidable. A bounded number of threads still admits straightforward encoding as we will see in Section 5.2.

5.2 Combining tree barrier

This section shows the verification in Jahob of the combining tree barrier, described by Scott and Mellor-Crummey [46].

I. Algorithm.
In the combining tree barrier each thread is initially associated with a distinct leaf of a common tree, e.g.:



In the tree each node has at most two children. An execution of the whole barrier consists of the synchronization phase and the wake-up phase. In the synchronization phase, threads start walking from their leaves towards the root (made precise below) such that each thread eventually begins waiting at a distinct node and such that at most one thread reaches the root. It will be guaranteed that a thread reaches the root only if all the other threads have started the computation. In the wake-up phase, the thread pointing to the root initiates a wake-up process (made precise below) in which every thread stops waiting and walks towards a leaf (which may be different from the leaf it has started with).

Each node of the tree carries the following information:

```
typedef struct node {
  node* parent, left, right; // these fields span the tree.
  int k; // constant, denotes the number of children.
  int count; /* initially k, used in synchronization, denotes the
                maximal number of threads that may still arrive at this node. */
  bool locksense; /* initially false, used for the wake-up, namely turns true when wake-up is signaled. */
  bool from_left; /* initially false, used for synchronization, namely
                     turns true when a thread arrives from the left. */
  bool from_right; /* initially false, used for synchronization, namely
                      turns true when a thread arrives from the right. */
} node;
```

If a node has no parent, left or right child, the corresponding fields have value 0. Notice that a node may have exactly one child, even long chains of one-child nodes are possible.

Each thread has, in addition to its program counter, two private variables:

```
local bool sense; // initially true, enables calling the barrier again.
local node* n; // initially a pointer to a distinct tree leaf.
```

An aside on sense should be made. We verify the first usage of the barrier right after initialization. The variable sense is needed for the future calls of the barrier function. This variable is still present for being as close to the original code as the current verification technology allows.

Each thread executes the following function:

```
void barrier() { // The synchronization phase starts.
A:  if(!(n→parent)) goto D; // if the current thread has already arrived at the root,
    // it may immediately proceed to the wake-up phase.
B:  // Otherwise there is a parent. Tell the parent whether the current thread
    // is reaching it from left or from right and go to the parent:
    if(n == (n→parent→left)) n→parent→from_left = true; else n→parent→from_right = true;
    n = n→parent;
C:  // Is this thread the first one to arrive at the node, or did some other thread arrive at the node earlier?
    // If this thread is the first one, prepare for waiting at the node, otherwise prepare for going to the parent:
    if((n→count)−− == 1) goto A; else goto F;
D:  // Wake-up phase.
    n→count = n→k; // reinitialize n→count to the number of children.
E:  // Signal wake-up to a possible thread waiting at node n:
    n→locksense = !(n→locksense); // also make the current thread skip waiting.
F:  // Waiting location. As long as no thread coming from the parent
    // signals wake-up, locksense remains false and sense remains true:
    while(sense!=(n→locksense));
G:  // If during the synchronization phase some thread came from the right, follow the right sub-tree.
    // Otherwise, if some thread came from the left, follow the left sub-tree:
    if(n→from_right) { n→from_right = false; n = n→right; goto D; }
    else if(n→from_left) { n→from_left = false; n = n→left; goto D; }
H:  sense = !sense; // arrived at a leaf; prepare sense for the next barrier call.
I:  }
```

The transitions at location `C` are crucial: they decrement the number of threads which may potentially arrive at the node `n`. If the previous value was greater than one, then one more thread has to arrive, and the current thread starts waiting for the wake-up. Otherwise the previous value is one, which means that if any other thread could arrive at node `n`, it has already done so; in this case all threads (except the thread executing the transition) coming from descendant leaves of `n` are waiting.

In our small example, the common parent p of the leaves of threads 0 and 1 starts with $p{\to}\mathsf{count} = 2$. When some thread arrives at p first, it decrements $p{\to}\mathsf{count}$ to 1 and starts waiting for the wake-up. The next thread that arrives at p diminishes $p{\to}\mathsf{count}$ to 0 and, noticing the fact that no more threads are going to arrive at p, proceeds to the parent of p. Notice that in our picture, thread 2 decrements the counters of the nodes between the initial leaf of thread 2 and the root (excluding both) from 1 to 0 without waiting.

We will consider the multithreaded program in which each thread starts at location `A` and cannot proceed beyond location `I`. Moreover, the transitions between each pair of consecutive control flow locations are assumed to be atomic.

As we will see, automation of the verification will require bounding the number of threads, but not any other quantity. Particularly, we assume that the synchronization tree has a fixed number of leaves, but the tree can be arbitrarily deep. Arbitrarily deep trees certainly subsume fixed-depth trees. In addition, they are useful for certain practical cases, namely when all threads except a fixed number are turned off, the nonparticipating leaves get removed from the tree together with the dead subtrees, but the remaining tree retains its structure.

II. Required reasoning.
The barrier property we verify is: when some thread arrives at location `I`, all threads are no more at location `A`. We will show what logic is needed to express an inductive invariant that is strong enough to prove this property.

To ensure the barrier property, it suffices to show that in the synchronization phase, if a thread arrives at the root, all other threads have started waiting. To make this property inductive, we need to characterize all nodes (and not just the root). We will now give a suitable characterization of the nodes in the synchronization phase. Let T be the set of non-null addresses of nodes in the tree and *Tid* the set of threads. Let

$$J \;=\; (\forall\, m\in T:\, m{\to}\mathsf{count} \geq S_1+S_2+S_3)\ ,$$

where the summands on the right hand side of the inequality are

$$\begin{aligned} S_1 &= |\{t\in Tid \mid n_t = m \wedge pc_t = \mathtt{C}\}|\,,\\ S_2 &= |\{\overline{m}\in T \mid \overline{m}{\to}\mathsf{parent} = m \wedge \overline{m}{\to}\mathsf{count} > 0\}|\,,\\ S_3 &= |\{t\in Tid \mid n_t{\to}\mathsf{parent} = m \wedge pc_t \in \{\mathtt{A},\mathtt{B}\}\}|\,, \end{aligned}$$

where n_t is the value of the private variable `n` of a thread t and pc_t is the control flow location of t.

For all reachable states in the synchronization phase J holds (proven by the verifier). In an initial state, for each node m we have: $S_1 = 0$, S_2 is the number of children that are internal nodes, and S_3 is the number of children that are leaves.

Now let us look at how the sum changes during transitions of the threads. If a thread goes from location `A` to location `B`, for all nodes the summands stay the same. If a thread goes from `B` to `C`, following a parent link, then for the child the summands stay the same, while for the parent S_1 increases by 1 and S_3 drops by 1. If a thread goes from `C` to `A`, then for the node to which the thread points S_1 drops by one, other summands stay the same, and the left hand side of the inequality drops by 1; for the parent of the node to which the thread points (if non-root) S_2 drops by 1 and S_3 increases by 1. The transition from `A` to `D` is not considered: it quits the synchronization phase and enters the wake-up phase. If a thread goes from `C` to `F`, then for the node to which the thread points S_1 drops by one, other summands stay the same, and the left hand side of the inequality drops by 1; for the parent of the node to which the thread points (if non-root) all the summands stay the same, especially S_2, since count of the child drops from 2 to $1 > 0$.

The property J can be written in a decidable logic by noticing that the presented algorithm exhibits only runs in which $0 \le m \rightarrow \mathsf{count} \le 2$ (since each node has at most $\mathsf{k} \le 2$ children). So two bits suffice to represent count (the same holds for k). For readability, we do not check these bounds, although this could be done, for instance, by using equality instead of inequality in J and asserting that the bit-pattern for count is never 11. The inequality $m \rightarrow \mathsf{count} \ge S_1 + S_2 + S_3$ can be written as

$$\begin{gathered}(S_1 = S_2 = S_3 = 0) \lor (m \rightarrow \mathsf{count} \neq 0 \land S_1 + S_2 + S_3 = 1) \lor \\ ((m \rightarrow \mathsf{count} = 2 \lor m \rightarrow \mathsf{count} = 3) \land S_1 + S_2 + S_3 = 2)\,.\end{gathered}$$

In turn, a formula of the form $S_1 + S_2 + S_3 = l$ can be expressed as a disjunction over all partitions of the number l, e.g. $S_1 + S_2 + S_3 = 2$ is equivalent to

$$\begin{gathered}(S_1 = 2 \land S_2 = S_3 = 0) \lor (S_2 = 2 \land S_1 = S_3 = 0) \lor (S_3 = 2 \land S_1 = S_2 = 0) \\ \lor (S_1 = S_2 = 1 \land S_3 = 0) \lor (S_1 = S_3 = 1 \land S_2 = 0) \lor (S_2 = S_3 = 1 \land S_1 = 0)\,.\end{gathered}$$

Expressing that a set has exactly zero, one or two elements can be done by using quantifiers. For instance, $|\{x \mid \phi(x)\}| = 2$ is equivalent to $\exists x_1, x_2\colon (x_1 \neq x_2 \land \phi(x_1) \land \phi(x_2) \land (\forall y\colon (\phi(y) \Rightarrow (y = x_1 \lor y = x_2))))$. For a fixed number of threads, quantifiers over threads turn into finite conjunctions and disjunctions.

Thus, for a fixed number of threads, J requires just the reasoning about the child-parent relationship in the trees and Boolean fields in the tree nodes. The logic that supports this reasoning is WS2S, the second-order logic of two successors.

III. Inductive invariant.

First we will describe different subformulas of the inductive invariant and then show the inductive invariant itself.

In the following, I_{st} will describe the local states of the threads in the synchronization phase. The formula says that all the threads are pointing into the tree (and not elsewhere, say, to 0) and are going up the tree or waiting. In addition, it says that no two different threads t, $\bar{t}$ can look at or use the parent pointer of the same node.

$$I_{\mathrm{st}} = \left(\forall t \in Tid\colon \begin{pmatrix} pc_t \in \{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{F}\} \land n_t \in T \\ \land \\ (pc_t \in \{\mathtt{A},\mathtt{B}\} \Rightarrow (n_t \rightarrow \mathsf{count} = 0 \land (\forall \bar{t} \in Tid \setminus \{t\}\colon (pc_{\bar{t}} \in \{\mathtt{A},\mathtt{B}\} \Rightarrow n_{\bar{t}} \neq n_t)))) \end{pmatrix}\right)\,.$$

The formula I_{sm} describes the tree in the synchronization phase. It states that J holds, that the wake-up is not yet signaled, and that if $\mathsf{from\_left}$ (resp. $\mathsf{from\_right}$) of a node is set, its left (resp. right) child exists.

$$I_{\mathrm{sm}} = \left(J \land \forall m \in T\colon \begin{pmatrix} \neg(m \rightarrow \mathsf{locksense}) \land \\ (m \rightarrow \mathsf{from\_left} \Rightarrow m \rightarrow \mathsf{left} \neq 0) \land (m \rightarrow \mathsf{from\_right} \Rightarrow m \rightarrow \mathsf{right} \neq 0) \end{pmatrix}\right)\,.$$

The formula I_{wt} describes the local states of the threads in the wake-up phase. The formula says that all the threads are pointing into the tree and are going down the tree or waiting.

$$I_{\mathrm{wt}} = (\forall t \in Tid\colon pc_t \in \{\mathtt{D},\mathtt{E},\mathtt{F},\mathtt{G},\mathtt{H},\mathtt{I}\} \land n_t \in T)\,.$$

The formula I_{wm} describes the tree in the wake-up phase. The formula says that if from_left (resp. from_right) of a node is set, its left (resp. right) child exists.

$$I_{\mathrm{wm}} = (\forall m \in T: \ ((m \rightarrow \mathsf{from\_left} \Rightarrow m \rightarrow \mathsf{left} \neq 0) \wedge (m \rightarrow \mathsf{from\_right} \Rightarrow m \rightarrow \mathsf{right} \neq 0)) \ .$$

The inductive invariant is $I = ((I_{\mathrm{st}} \wedge I_{\mathrm{sm}}) \vee (I_{\mathrm{wt}} \wedge I_{\mathrm{wm}}))$.

IV. Encoded program and its invariant.
We have chosen Jahob (with its back-end MONA for handling WS2S) to encode and verify the combining tree barrier.

In the encoding (written in Java syntax) each node carries the following information:

```
class Node {
  boolean count_bit_1, count_bit_0, k_bit_1, k_bit_0;
  Node parent, left, right;
  boolean locksense, from_left, from_right;
}
```

The program counter of each thread t is encoded by four bits pc$t$_bit_3, pc$t$_bit_2, pc$t$_bit_1, pc$t$_bit_0 as follows: A=0000, B=0001, C=0010, D=0011, E=0100, F=0101, G=0110, H=0111, I=1000. For each thread t there is a variable nt pointing into the tree (with initial value leaf$t$) and a Boolean sense$t$.

We encode the multithreaded program in Jahob as a nondeterministic program:

```
// The scheduler loop:
while(true) /*: invariant  ... see above ... */ {
  if(nondeterministic choice) { // transitions of thread 1
    ...
  } else if(nondeterministic choice) { // transitions of thread 2
    ...
  } else { // transitions of thread 3
    ...
  } // end of choosing the thread.
} // end of the scheduler loop.
```

Shortly we will show the transitions of the first thread. The transitions are organized into structuring blocks {...} of the programming language syntax. A correct block in an execution is chosen by nested **if-then-else**s depending on the values of the quadruple of variables (pc1_bit_3, pc1_bit_2, pc1_bit_1, pc1_bit_0). Inside a block, the transition at the control flow location encoded by the quadruple is executed on the variables pc1_bit_i ($0 \leq i \leq 3$), sense1 and the fields of n1. Each block itself is a direct translation of the corresponding original transition from page 25 for thread 1. For example, consider the following command:

```
A:  if(!(n→parent)) goto D;
B: ...
```

Since A is encoded as 0000, B as 0001 and D as 0011, the translation is

```
if(n1.parent==null) pc1_bit_1=true;
pc1_bit_0=true;
```

Other transitions of thread 1 are translated likewise. The full translation of thread 1 is given in Fig. 1. Transitions of the other threads are encoded analogously.

V. Using the tool.
Jahob allowed verification for two and three threads. It is possible for an internal node to have exactly one child, thus the tree might be arbitrarily deep. We will come to this point

```
if(pc1_bit_3) { /* A thread starting from location I stays at I. */ }
else {
  if(pc1_bit_2) {
    if(pc1_bit_1) {
      if(pc1_bit_0) { // modeling transition H→I
        sense1=!sense1; pc1_bit_3=true; pc1_bit_2=pc1_bit_1=pc1_bit_0=false;
      } else { // !pc1_bit_0, modeling transitions G→D, G→H
        if(n1.from_right) { n1.from_right=false; n1=n1.right; pc1_bit_2=false; /* goto D */ }
        else if(n1.from_left) { n1.from_left=false; n1=n1.left; pc1_bit_2=false; /* goto D */ }
        pc1_bit_0=true; // else goto H
      }
    } else { // !pc1_bit_1
      if(pc1_bit_0) { // modeling transition F→G
        if(sense1==n1.locksense) { pc1_bit_1=true; pc1_bit_0=false; }
      } else { // modeling transition E→F
        n1.locksense=!(n1.locksense); pc1_bit_0=true; // goto F
      }
    } // end of if(pc1_bit_1)-else
  } else { // !pc1_bit_2
    if(pc1_bit_1) {
      if(pc1_bit_0) { // modeling transition D→E
        n1.count_bit_1=n1.k_bit_1; n1.count_bit_0=n1.k_bit_0;
        pc1_bit_2=true; pc1_bit_1=pc1_bit_0=false; // goto E
      } else { // !pc1_bit_0, modeling transitions C→A, C→F
        if(n1.count_bit_0) n1.count_bit_0=false;
        else { n1.count_bit_0=true; n1.count_bit_1=!(n1.count_bit_1); }
        if(n1.count_bit_0 || n1.count_bit_1) { pc1_bit_2=pc1_bit_0=true; pc1_bit_1=false; /*goto F*/}
        else pc1_bit_1=false; // goto A
      }
    } else { // !pc1_bit_1
      if(pc1_bit_0) { // modeling transition B→C
        if(n1==n1.parent.left) n1.parent.from_left=true; else n1.parent.from_right=true;
        n1=n1.parent; pc1_bit_1=true; pc1_bit_0=false; // goto C
      } else { // modeling transitions A→B, A→D
        if(n1.parent==null) pc1_bit_1=true; // goto D else goto B
        pc1_bit_0=true;
      } // end of if(pc1_bit_0)-else
    } // end of if(pc1_bit_1)-else
  } // end of if(pc1_bit_2)-else
} // end of if(pc1_bit_3)-else
```

Fig. 1 Encoding of transitions of the first thread

later, noticing for now that for real software barriers such a situation can occur when a barrier tree is designed and fixed for an arbitrary number of threads while only a bounded number of threads participate in a barrier call, causing many tree branches to be cut.

We had to improve Jahob to verify the combining tree barrier:

- If MONA failed to prove that a formula in WS2S is valid, the reason why was not visible. MONA could have failed to start, or it could have run out of space, or the formula could just have been invalid. We have improved the interaction between Jahob and MONA so that appropriate countermeasures could be taken in each case.
- The translation of the subformula J of the invariant into WS2S is large. That led to an overflow during formula printing. We have repaired this issue.
- Thomas Wies has provided a shorter definition of the null pointer in WS2S for better scalability and repaired the (previously broken) translation of negation into WS2S.

- Previous splitting in Jahob was not sufficient. It was done only in the consequent of a sequent: after splitting $A \Rightarrow (C \wedge D)$ into $A \Rightarrow C$ and $A \Rightarrow D$, MONA could run out of space on, say, $A \Rightarrow C$, just because the input formula is large. Our improvement is to continue splitting recursively on arbitrarily deep disjunctions inside the antecedent if MONA runs out of space: $(\phi \wedge (A \vee B) \wedge \psi) \Rightarrow C$ now can be split on need into $(\phi \wedge A \wedge \psi) \Rightarrow C$ and $(\phi \wedge B \wedge \psi) \Rightarrow C$. If more than the theorem prover MONA were necessary (say, if we had required Z3 additionally), sub-goals could be handed over to different theorem provers.

Given the above improvements, two threads could be verified in 67 s, and three threads could be verified in 1721 s on an Intel® Core™ i5-520M CPU clocked at 2.4 GHz with 8 GB RAM clocked at 1066 MHz.

VI. Beyond the frontier.
An attempt to verify four threads failed: the input for MONA gets too large, even after all potentially possible case splitting is done. To handle such cases theorem provers for WS2S must be improved or a new decision procedure must be invented to directly encode the formulas like J. These improvements are beyond the scope of the present paper.

An alternative line of work would be an encoding of the arbitrary-threaded barrier into a program whose proof could be expressed in WS2S. Such an encoding (similar to that of the carry-lookahead adder [3]) is indirect and would require verification itself.

5.3 Static tree barrier vs. Combining tree barrier

Now we evaluate our experience in the verification of tree barriers.

In both barriers each execution can be separated into the synchronization and wake-up phases. The inductive invariant is $I_{\mathrm{s}} \vee I_{\mathrm{w}}$ where I_{s} is the invariant for the synchronization phase and I_{w} is the invariant for the wake-up phase.

For both barriers I_{s} can be written as a conjunction $I_{\mathrm{st}} \wedge I_{\mathrm{sm}}$ where I_{st} speaks about all threads and I_{sm} speaks about all tree nodes. For the static tree barrier, both I_{st} and I_{sm} are of the form $(\forall t \in T\colon \ldots)$, since threads are associated with nodes. The combining tree barrier enjoys a separate notion of threads, which makes I_{st} and I_{sm} have the forms $(\forall t \in \mathit{Tid}\colon \ldots)$ and $(\forall n \in T\colon \ldots)$, respectively.

For both barriers I_{w} can also be written as a conjunction $I_{\mathrm{wt}} \wedge I_{\mathrm{wm}}$ where I_{wt} speaks about all the threads and I_{wm} speaks about all the tree nodes. For the static tree barrier, I_{wt} is of the form $(\forall t \in T\colon \ldots)$, since threads are associated with nodes, and $I_{\mathrm{wm}} = \mathit{true}$. For the combining tree barrier, I_{wt} is of the form $(\forall t \in \mathit{Tid}\colon \ldots)$ and I_{wm} is still of the form $(\forall n \in T\colon \ldots)$, but very weak.

In both barriers I_{s} is relatively strong and I_{w} relatively weak compared to the strongest inductive invariant for the corresponding phase.

For both barriers the inductive invariants with the code were fed into Jahob, converted to WS2S and verified by MONA.

Based on these experiences, we expect that user-given variations of the tree-based barriers can also be handled in Jahob similarly.

6 Array-based barriers

There are best-case-logarithmic barriers that do not resort to a link-based tree data structure. Instead, threads distribute the information through a shared array such that each cell of the array is accessed by a constant number of threads. We will verify two most renowned representatives of array-based barriers: the dissemination and the tournament barriers.

6.1 Dissemination barrier.

We begin with an algorithm for the dissemination barrier due to Hensgen et al. [19].

I. Algorithm.
First we informally describe the algorithm, then show its code and finally display an example run of the algorithm.

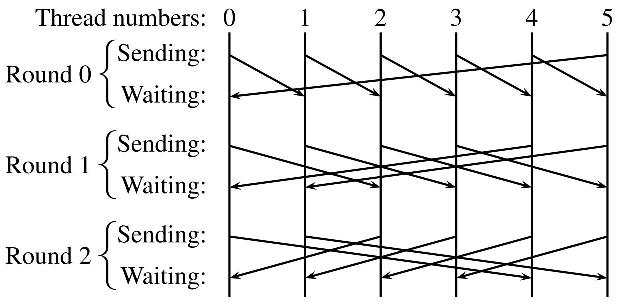
Let n participating threads be identified by numbers from 0 till $n-1$. In an execution of the dissemination barrier a thread executes $L = \lceil \log_2 n \rceil$ synchronization rounds. In round i, thread number t sends a signal to thread number $(t+2^i) \bmod n$ (where $\bmod n$ gives the smallest nonnegative remainder after division by n) and starts waiting for a signal from any thread. The thread that will send this signal is $(t-2^i) \bmod n$. It turns out that after L rounds, each thread x has received a signal from every other thread y, either directly or transitively, i.e. some thread z has received a signal (directly or transitively) from y and sent its signal to x.

Signals are stored in an array A: ((thread identifier)$\times$(round number))$\to$bool.

```
shared bool A[n][L]; // the array entries are initially false.
// Each thread t executes the following function:
void barrier() {
  for(int round=0; round<L; round++) {
    A[(t+2^round)%n][round] = true; // send a signal to thread (t+2^round) mod n.
    // Wait for a signal from thread (t−2^round) mod n:
    await A[t][round]; // implemented, e.g. as a loop.
  }
}
```

Powers of 2 can be constructed by a left shift or precomputed during initialization.

For example, 6 threads send signals and wait for them as follows:



We will consider the multithreaded program in which each thread starts at the head of its **for** loop with round = 0 and stops right after exiting its **for** loop.

II. Required reasoning.
We need to reason about two-dimensional arrays, since we want to express facts about A. Also we need to reason about powers of two and arithmetic modulo n. Luckily, we do not need all the properties of the power or of the remainder. With respect to the powers of 2, our experiments show that the proof of the barrier property just requires the property

$n \leq 2^L < 2n$; it is possible to provide this fact to the theorem prover as an axiom. With respect to the remainder after division by n, the proof relies on the exact values of remainder only on a bounded interval $[-3n, 3n)$. We characterize the remainder as follows:

$$\forall a \in \mathbb{Z}\colon \begin{pmatrix} (0 \leq a \operatorname{mod} n < n) \\ \wedge\ (\ -3n \leq a < -2n \Rightarrow a \operatorname{mod} n = a + 3n) \\ \wedge\ (\ -2n \leq a < -n \Rightarrow a \operatorname{mod} n = a + 2n) \\ \vdots \\ \wedge\ (\ 2n \leq a < 3n \Rightarrow a \operatorname{mod} n = a - 2n) \end{pmatrix}.$$

On $\mathbb{Z} \setminus [-3n, 3n)$, our remainder abstraction is less restricted.

As a result, we just need to reason about two-dimensional arrays and linear integer arithmetic. This fragment is still undecidable, but modern theorem provers employ heuristics that allow proving validity of many formulas from this fragment in practice.

III. Inductive invariant.
For the algorithm above, the barrier property we verify is: if some thread exits the barrier function, then any other thread is either not at the loop head or has a positive value of round (or both). For expressing this, let $round_t$ (resp. pc_t) be the value of the variable round (resp. of the control flow location) of a thread t. Notice that for any reachable state of the barrier and any thread t all of the following four facts hold (loosely formulated).

1. $round_t \in [0, L]$;
2. Threads $\hat{t}$ that have a number which is less than t by a certain amount (made precise below) and that are not at the waiting location have transitively sent a signal to t and thus have executed their loop at least once.
3. If $\mathsf{A}[t][l]$ is true for some l, then all the threads $\hat{t}$ that have a number less than t by a certain amount (made precise below) and that are not at the waiting location have transitively sent a signal to t and thus have executed their loop at least once.
4. One of the following facts holds:
   - t is at the head of the **for** loop, or
   - t is at the waiting location, $\mathsf{A}[(t + 2^{round_t}) \operatorname{mod} n][round_t]$ is true (since t has previously made it true) and the guard condition holds ($round_t < L$), or
   - t has exited the loop and $round_t \geq L$.

The invariant I is the conjunction of 1.-4. For the sake of completeness, we will now fully specify the parts 2 and 3 (proved by the verifier):
A thread $\hat{t}$ that is at the head of the loop or at the end of the barrier function satisfies $round_{\hat{t}} > 0$ if any of the following conditions 2. or 3. hold.

2. $0 < ((t - \hat{t}) \operatorname{mod} n) < 2^{round_t}$, or
3. $\mathsf{A}[t][l]$ is true and $((t - 2^l - \hat{t}) \operatorname{mod} n) < 2^l$ for some l.

IV. Encoded program and its invariant.
We have chosen Boogie as the language to encode the dissemination barrier. Boogie does not support concurrency, though it supports nondeterminism. We encode the multithreaded program as a nondeterministic sequential one.

It turns out that the tool chain Boogie+Z3 runs out of space if the program is supplied as a monolithic chunk as in Section 5 (the formulas that Z3 has to digest are too complex to be proven). Thus, we partition the program's transitions and let Boogie check them separately:

- Transitions APL$\rightarrow$BPL for some thread t,
- Transitions BPL$\rightarrow$APL for some thread t,

– Transitions APL→CPL for some thread t.

Here, APL is the loop head, BPL is the waiting location and CPL is the location after the loop. We encode each such block of transitions as a procedure that has the above inductive invariant as a pre- and a post-condition. Procedures are checked separately: in our case, Boogie does not run out of space on them. Let us consider, for example, the procedure for transitions from BPL to APL:

```
procedure procBPL2APL()
requires Inv(...); // when a transition BPL→APL starts, assume that the invariant holds.
modifies pc, round; // modifies the control flow of some thread and a round number, but not A.
ensures Inv(...); // when the transition finishes, Boogie has to prove the invariant.
{ var t : int; // t is the thread that executes the transition.
  var nextRound, nextPc; // local to this procedure.
  // Assume that a thread t starts at BPL and that the signal is received:
  assume 0≤t ∧ t<n ∧ pc[t]=BPL ∧ A[t][round[t]];
  nextRound :=round[t↦(round[t]+1)]; // construct the map containing the next round numbers.
  nextPc :=pc[t↦APL]; // construct the map containing the next control flow locations.
  // Help Boogie in proving the four parts of the invariant (explained below):
  call forall Lemma1(t,*,nextPc,nextRound); // These Lemmas are
  ...                                        // discharged in other
  call forall Lemma4(t,*,nextPc,nextRound); // procedures (shown online).
  // Update the maps shared between procedures responsible for transitions APL→BPL, APL→CPL, BPL→APL:
  round :=nextRound; pc :=nextPc;
}
```

In the above, the notation m[x↦y] means a map which is identical to the map m for all arguments expect the argument x for which it returns the value y.

The current state of the art of theorem proving still requires a little help from the user. The help comes in form of procedures that concentrate the theorem prover on separately proving parts of the invariant. The body of those procedures is empty, the precondition, say, of Lemma1, contains only the information that is necessary to prove part 1 of the invariant, the postcondition is part 1 itself. This scheme allows the theorem prover to prove part 1. Without those procedures Z3 would have too many assumptions to choose from for proving part 1 and would fail in choosing the right assumptions. Without going into details, the statement "**call forall** Lemma1(t,*,nextPc,nextRound)" is equivalent to assuming the formula

$$\forall t' : \begin{pmatrix} \text{precondition of } \mathsf{Lemma1}(\mathsf{t}, t', \mathsf{nextPc}, \mathsf{nextRound}) \Rightarrow \\ \text{postcondition of } \mathsf{Lemma1}(\mathsf{t}, t', \mathsf{nextPc}, \mathsf{nextRound}) \end{pmatrix},$$

while writing the body of Lemma1 ensures that this formula is proven valid. For further description of the **call forall** statement we refer the interested reader to the manual of Boogie 2 [29].

V. Using the tool.

The supporting lemmas and the inductive invariant can be proven in altogether 1.03 s on an Intel® Core™ i5-3320M CPU clocked at 2.6 GHz with 8 GB RAM clocked at 1600 MHz. During experiments with Boogie, we have found that trying to prove a program property which does not hold results in an almost immediate rejection, which is a pleasant help during debugging.

VI. Beyond the frontier.

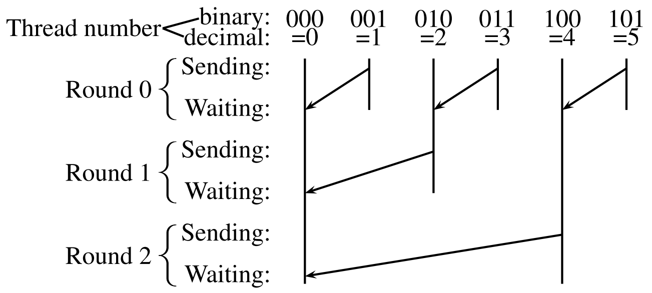
The current state of the art of theorem proving requires supporting lemmas. Due to undecidability of the logic that Boogie syntactically accepts, we expect that the tool will also require manual support in the future. Reducing the amount of manual support would be the next step in increasing the automation of proving the correctness of the dissemination barrier.

6.2 Tournament barrier

This subsection considers the verification of the tournament algorithm with tree-based wake-up, given in `Bfly1/tournament.c` of [36] and explained by Scott and Mellor-Crummey [46] and Hensgen et al. [19].

I. Algorithm.
First we informally describe the tournament algorithm, following which we show its code. We support the presentation of both by a running example.

A computation of the tournament barrier is akin to a chess tournament: in each round, threads are partitioned into couples, in each couple one of the threads wins and proceeds to the next round, the other thread loses and starts waiting until the winner comes back and wakes the thread up. If the number of threads is not a power of two, threads without opponents skip certain rounds. The difference to the real tournament is that the winner and the loser, as well as the couplings, are statically determined by the bits of the thread identifiers (a nonnegative integer represented by a bitstring). For example, six threads will synchronize as follows:



After the champion (here: 000) has arrived at the last round, it will wake up every thread that has lost to the champion directly during the synchronization phase (here: 100, 010 and 001). Once a thread is awake, it will wake up all the threads that have lost to it directly.

The general win/lose rule is the following.

Each identifier is of the form $x0\ldots0$ with $i \geq 0$ zeros after the bitstring x. The thread with such an identifier will win in each round $j \in [0,i)$ over the thread $x0\ldots010\ldots0$ where the rightmost 1 is on the jth position from the right. If $x0\ldots010\ldots0$ does not correspond to a thread, $x\underbrace{0\ldots0}_{i\text{ zeros}}$ will skip the jth round. Also, if a thread is of the form $x1\underbrace{0\ldots0}_{j\text{ zeros}}$ for some bitstring x and some $j \geq 0$, then the thread will lose in round j to the thread $x0\underbrace{0\ldots0}_{j\text{ zeros}}$. Thread $0\ldots0$ is the champion: it will win in each round.

When a thread is going to win in round i, it waits for the loser to reach that round before proceeding. If a thread is going to lose in round i, it arrives there, sends a signal to the winner and starts waiting for the wake-up signal. All the signals get stored in a two-dimensional array A: ((thread identifier)$\times$(round number))$\to$bool.

In the following code of the tournament barrier, $n \geq 2$ is the number of threads and $L = \lceil \log_2 n \rceil$ the number of rounds.

```
shared bool A[n][L]; // all array entries are initially false.
local int tid; // unique thread identifier in [0,n).
// Each thread calls the following function:
void barrier() { // Start the synchronization phase.
```

```
  int round = 0; // the current round number.
  // Participate in the tournament as long as the thread does not lose:
  for( ; round<$L$−1 && (tid%$2^{\text{round}+1}$ == 0) ; round++ )
    // Win the round or skip the round:
    if(tid+$2^{\text{round}}$<$n$) await A[tid][round]; // if the opponent (loser) exists, wait for the loser.
  if((tid%($2^{\text{round}+1}$)) $\neq$ 0) { // the thread is losing this round.
    A[tid−$2^{\text{round}}$][round] = true; // inform the opponent (winner).
    await A[tid][round]; // wait for the wake-up from the winner.
  } else { // last round $L-1$, the thread is the champion.
    await A[tid][round]; // wait for the loser.
    A[tid+$2^{\text{round}}$][round] = true; // wake up the loser.
  } // end of if((tid%($2^{\text{round}+1}$)) $\neq$ 0).
  // Now the thread is awake. Either it is a loser of the current round and the winner has woken up the thread.
  // Or it is the champion and it has woken up the loser. In any case, the opponent is awake.
  // The thread should wake up the losers of the previous rounds.
  for( ; round>1 ; round−−) { // go through all rounds > 1
    if(tid%($2^{\text{round}}$)==0 && tid+$2^{\text{round}-1}$<$n$) // if the thread was the winner of the round
      // and if its opponent existed,
      A[tid+$2^{\text{round}-1}$][round−1] = true; // wake up the loser.
  } // end for.
  // Wake up the loser of the initial round, if necessary:
  if(round==1 && (tid is even) && t+1<$n$) A[tid+1][0] = true;
}
```

For our six-thread example, thread 010 would start the 0th round by going into the body of the first loop, where it waits for a signal (notice that only thread 011 may send it). After getting the signal, it exits the first loop, goes into the if-part of the conditional statement after the loop, sends the signal to 000 and starts waiting for the wake-up.

At this moment, round=1 for thread 010. When 010 gets awoken by the champion, it skips the second **for** loop, raises A[011][0] to wake up the thread 011 and exits. (As we will see, it is convenient to consider the last iteration of the loop separately, therefore we have taken it out of the wake-up loop.)

The above code can be optimized, e.g. to avoid explicit computations of powers of two. (But beware that optimizations increase the possibility of bugs—we have even found one such bug in `Bfly1/tournament.c` of [36] and repaired it.)

We consider the multithreaded program in which each thread executes the body of the barrier function.

II. Required reasoning.
We now explain why we require certain nontrivial features in a correctness proof and how we deal with them.

Syntactically, the program contains two-dimensional arrays, powers of two and remainders modulo a power of two. Such powers and remainders can be relatively simply carried out on the bitwise representation of the numbers. To reason about a bitwise representation of a number we need unbounded bit arrays.

Thus we handle thread identifiers as unbounded arrays of bits. More concretely, we model the set of identifiers by a set T of bit arrays such that both of the following conditions hold (from now on we view L as an ordinal, i.e. as the set of all smaller ordinals).

– The support of an array in T is a subset of L, i.e. if a bit in an array in T is raised, the index of the bit is in L.
– If an array a in T represents a loser for some round i, then the identifiers of the transitive winners over a are in T. (A transitive winner over a is either the direct winner over a or a transitive winner over the direct winner.)

The set $\{t \in L \to \{0,1\} \mid \sum_{i<L} t(i) \cdot 2^i < n\}$ representing the integer interval $[0,n)$, which is used in the original tournament algorithm [19], satisfies the aforesaid conditions, but there are more sets that satisfy these conditions, e.g. $\{000, 001, 010, 100, 101\}$. Thus we support richer sets of thread identifiers and are able to handle more inputs. The overhead for handling more inputs turned out to be negligible.

Due to substituting thread identifiers by bit arrays, $\mathsf{A} \in ((T \times L) \to \text{bool})$ gets a second-order map: its first index is a map itself.

Thus, we require a logic which can express integers, first-order maps and two-argument second-order maps. This fragment is undecidable, but heuristics of modern theorem provers allow proving validity of many formulas from this fragment in practice.

We also require maps as first-order objects, but theorem provers support only first-order arrays. There are two differences.

- Theorem provers allow comparing elements, but forbid comparison of maps (for reference: maps are called functions by the theorem proving community). If f and g are maps, $f = g$ is an invalid piece of syntax. Theorem prover clients cannot syntactically build a modified map $f[x \mapsto y]$ either. We can, however, take the value of a map for an argument, as in $f(x)$.
- Theorem provers allow syntactical comparison of arrays. If f and g are arrays, $f = g$ is a syntactically valid term. The value stored at an index x in an array f is "**select** $f\ x$", and a modified map $f[x \mapsto y]$ is "**store** $f\ x\ y$".

The known proof of the barrier property requires comparing and building modified thread identifiers. Thus, our thread identifiers have to be represented as arrays for the theorem prover. This capability is not present is Jahob, but it is present in Boogie.

We also overcame another difficulty pertaining to maps and arrays. By default, verifiers are using a relatively weak logic for arrays for the sake of termination or speed. In particular, the logic ignores the extensionality axiom which states that arrays f and g are equal if $(\forall x\colon f(x) = g(x))$. The proof of the barrier property turns out to require a form of the extensionality axiom, so we explicitly provide this axiom to the verifier.

III. Inductive invariant.

The barrier property we verify is: if some thread arrives at the final control flow location at the end of the barrier function, all other threads are not at the head of the first **for** loop and not before it. Now we will describe the high-level structure of an inductive invariant I that is sufficient to prove the barrier property.

For a state of the program, let $round_t$ denote the value of the variable `round`.

The invariant is a disjunction of two formulas: one for the synchronization phase and the other for the wake-up phase. The formula describing the synchronization phase is a conjunction of several facts similar to those from Section 6.1. We state two most important facts:

- Every thread is either in the **for** loop or the if-then part immediately following it (i.e. it is at the loop head, or has lost and is waiting for the wake-up, or has won not in the last round and is waiting for the loser, or it is the champion and is waiting for its opponent in the last round).
- If $\mathsf{A}[t][l]$ is true for some thread t and some $l < L$, then
  - t wins or skips all the rounds till (and including) round l and
  - any other thread $\hat{t}$ that loses to t transitively till (and including) round l is waiting for the wake-up. Here "transitively" means that $\hat{t}$ loses to t directly or loses to some other thread that loses to t transitively.

In the above description, winning and losing refer to bits in thread identifiers as described previously.

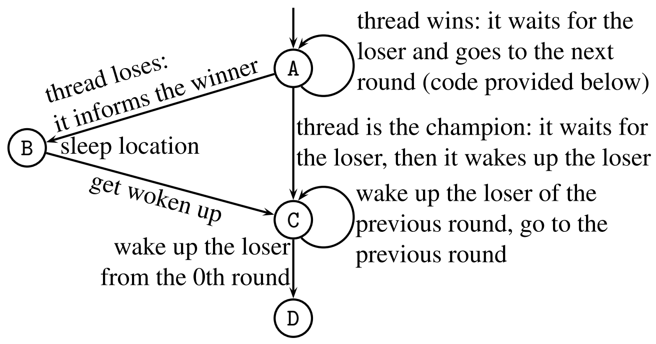
For example, if A[010][0] is true in the synchronization phase, then some thread raised this bit when it was in the if-part after the synchronization loop. The identifier of this thread is 011. After having raised that bit, 011 started waiting. It cannot be woken up in the synchronization phase (otherwise it would be woken up by thread 100 in its 0th round, which contradicts the if-condition $100\%(2^{round_{100}+1}) \neq 0$). Thus, in the synchronization phase, thread 011 continues waiting.

The formula describing the wake-up phase says that each thread is at a certain control flow location, either waiting for the wake-up or proceeding after wake-up.

If a thread is at the last control flow location, the barrier cannot be in the synchronization phase, so the whole barrier is in the wake-up phase, so no thread can be before or at the head of its first **for** loop.

IV. Encoded program and its invariant.
The required reasoning is supported by the Boogie+Z3 tool chain; we will explicate the real code and fully show the inductive invariant.

To stay on the positive side of the verifiability frontier, we produce a coarse-grained program with four control flow locations per thread: `A`, `B`, `C` and `D`. A thread moves between them as in the following diagram (details omitted):


thread wins: it waits for the loser and goes to the next round (code provided below)
thread loses: it informs the winner
sleep location
thread is the champion: it waits for the loser, then it wakes up the loser
get woken up
wake up the loser of the previous round, go to the previous round
wake up the loser from the 0th round
A
B
C
D

Now we will look, for example, at the transition $\mathtt{A} \rightarrow \mathtt{B}$. The transition's precondition for a thread t is a conjunction of two conditions:

- the thread should have won or skipped its previous rounds, i.e. $t(0) = \ldots = t(round_t - 1) = 0$, and
- the thread should lose the current round, i.e. $t(round_t) = 1$.

The effect of the transition is sending a signal to the opponent (winner $t[round_t \mapsto 0]$) by raising $\mathsf{A}[t[round_t \mapsto 0]][round_t]$.

With respect to the example above, for the thread 010 in a state with $round_t = 1$ the precondition is $t(0) = 0 \wedge t(1) = 1$ and the effect is $\mathsf{A}[000][1] := true$.

Given the above thread structure, we first construct parts of the formula describing the inductive invariant and then give the inductive invariant itself.

In the following formulas, an expression of the form $t|_{(a,b)}$ (resp. $t|_{[a,b)}$) denotes the restriction of the map t to the integer interval $\{x \in \operatorname{dom} t \mid a < x < b\}$ (resp. $\{x \in \operatorname{dom} t \mid a \leq x < b\}$).

The set of thread identifiers T which was sloppily described on p. 35 is formally characterized by the following formula:

$$I_{\mathrm{T}} \quad = \quad \big(\forall \hat{t} {\in} T, t {\in} (L {\to} bool) : \big(\big(\exists i {<} L\colon\ \big(t|_{(i,L)} {=} \hat{t}|_{(i,L)} \wedge \hat{t}(i) \wedge (\forall j {\leq} i\colon \neg t(j))\big)\big) \Rightarrow t \in T\big)\big)\ .$$

In the synchronization phase, the round numbers should be bounded by the binary logarithm of the number of threads:

$$I_{\mathrm{s}1}(t) \quad = \quad (0 \le \mathit{round}_t < L)\ .$$

Now we will mention control flow locations: for a state of the program, pc_t will be the program counter of thread t.

In the synchronization phase, as we have already seen, if $\mathsf{A}[t][l]$ is true for some thread t and some $l < L$, then

- t wins or skips all the rounds till (and including) round l and
- any other thread $\hat{t}$ that loses to t transitively till (and including) round l is waiting at location B.

Formally:

$$I_{\mathrm{s}2}(t) \ = \ \big(\forall l{\in}[0,L) : \big(\mathsf{A}[t][l] \Rightarrow \big((\forall i{\le}l\colon \neg t(i)) \wedge \big(\forall \hat{t}{\in}T\colon \big((\hat{t}(l) \wedge \hat{t}|_{(l,L)}{=}t|_{(l,L)}\big) \Rightarrow pc_{\hat{t}}{=}\mathtt{B}\big)\big)\big)\big)\ .$$

In the synchronization phase, if any two threads have the same win-lose behavior from the current round number of one thread onwards, then the other thread is waiting for the wake-up:

$$I_{\mathrm{s}3}(t) \quad = \quad \big(\forall \hat{t}{\in}T\setminus\{t\}\colon \ \big(\hat{t}|_{[\mathit{round}_t,L)}{=}t|_{[\mathit{round}_t,L)} \ \Rightarrow \ pc_{\hat{t}}{=}\mathtt{B}\big)\big)\ .$$

In the synchronization phase a thread can be either at location `A` or at location `B`. If a thread is at location `A` and has not received a signal for its current round, then it has been winning the previous rounds, and if in addition it is going to lose its current round to an opponent, then the opponent is still in the process of synchronization, i.e. at location `A`. If a thread is at location `B`, then its wake-up signal is not yet raised, it has lost its current round but won all previous rounds. So let

$$I_{\mathrm{s}4}(t) \quad = \quad \left(\begin{array}{l} \left(pc_t{=}\mathtt{A}\wedge \left(\begin{array}{l} \mathsf{A}[t][\mathit{round}_t] \vee \\ \left(\begin{array}{l} (\forall i{<}\mathit{round}_t :\ \neg t(i)) \ \wedge \\ \left(\left(t(\mathit{round}_t) \wedge t[\mathit{round}_t \mapsto 0]{\in}T\right) \Rightarrow pc_{t[\mathit{round}_t \mapsto 0]} = \mathtt{A}\right) \end{array}\right) \end{array}\right)\right) \\ \vee \\ (pc_t{=}\mathtt{B} \ \wedge \ \neg\mathsf{A}[t][\mathit{round}_t] \ \wedge \ t(\mathit{round}_t) \ \wedge \ (\forall i{<}\mathit{round}_t :\ \neg t(i))) \end{array}\right)\ .$$

The formula fully describing the synchronization phase declares that all the just described formulas hold for an arbitrary thread:

$$I_{\mathrm{s}} \quad = \quad (\forall t{\in}T :\ (I_{\mathrm{s}1}(t) \wedge I_{\mathrm{s}2}(t) \wedge I_{\mathrm{s}3}(t) \wedge I_{\mathrm{s}4}(t)))\ .$$

The formula describing the wake-up phase just asserts that all threads have left the control flow location `A`:

$$I_{\mathrm{w}} \quad = \quad (\forall t{\in}T :\ pc_t \in \{\mathtt{B},\mathtt{C},\mathtt{D}\})\ .$$

The inductive invariant declares how the thread set looks like and that the threads are either in the synchronization or in the wake-up phase:

$$I \quad = \quad (I_{\mathrm{T}} \wedge (I_{\mathrm{s}} \vee I_{\mathrm{w}}))\,.$$

As in the case of the dissemination barrier, the whole multithreaded program is too big to be verified monolithically. To overcome this limitation, we encode each transition in a separate procedure. For example, let us look at the procedure corresponding to moves `A`$\rightarrow$`A` of all the threads.

```
procedure A2A() // encoding the self-loop at A.
requires Inv(...); // assume that the invariant holds before the transition.
modifies round; // modifies just the round number. The variable "round" maps threads to their round numbers.
ensures Inv(...); // prove that the invariant holds after the transition.
{ var t: BitArray; // BitArray = int→bool, t is the thread that executes the transition.
  var nextRound: [BitArray]int; // "nextRound" is the round map after the transition.
  assume InT[t] // the identifier t should be valid, i.e. in the set T.
    ∧pc[t]=A // the program counter should be A.
    ∧ZeroPrefix(t,round[t]) // t[0]=...=t[round_t]=0, i.e. thread t wins or skips the rounds 0..round_t.
    ∧round[t]+1<L // the round is not the last one.
    (InT[t[round[t]↦true]] ⟹A[t][round[t]]) // if the opponent is valid, i.e. in T, then
        // the opponent should have signaled its arrival by setting the mentioned element of A to true.
  nextRound:=round[t↦(round[t]+1)]; // construct the next round numbers for all the threads.
  call forall Lemma...(t,∗,nextRound); // help in proving certain
  call forall Lemma...(t,∗,nextRound); // parts of the invariant.
  round :=nextRound; // update the current round numbers.
}
```

If the conjunction of the conditions in the **assume** statement is false, then the thread just blocks, which it should.

In the above code, the **call forall** statements help proving the invariant as in Section 6.1; we will not go into details here.

V. Using the tool.
Encoding the tournament barrier in Boogie requires reducing the operations and the reasoning about operations to concepts that the tool understands: Booleans, integers and arrays. After such an encoding is done, we found out that certain parts of the invariant cannot be proven fully automatically. After guiding the proof by the **call forall** statements, we were able to make the Boogie tool prove the invariant in 711.8 s on an Intel® Core™ i7-3720QM CPU clocked at 2.6 GHz with 8 GB RAM clocked at 1600 MHz.

VI. Beyond the frontiers.
During construction of the barrier we had to reduce the number of control flow locations per thread and provide guidance to the theorem prover. Permitting more locations per thread or removing the supporting lemmas would place the barrier on the other side of the verifiability frontier. Addressing these next challenges is out of scope of the present paper.

7 A central barrier with a client

This section investigates a client of a barrier together with the barrier. We consider an implementation of a barrier in .NET [39] and a variation of the publicly available MSDN example [40] that uses the implementation.

I. Algorithm.
First we show the client and then expound on the used barrier implementation.

The following client code creates four threads, each thread calls the barrier function (in .NET it is named SignalAndWait) four times and increments a shared variable between the calls.

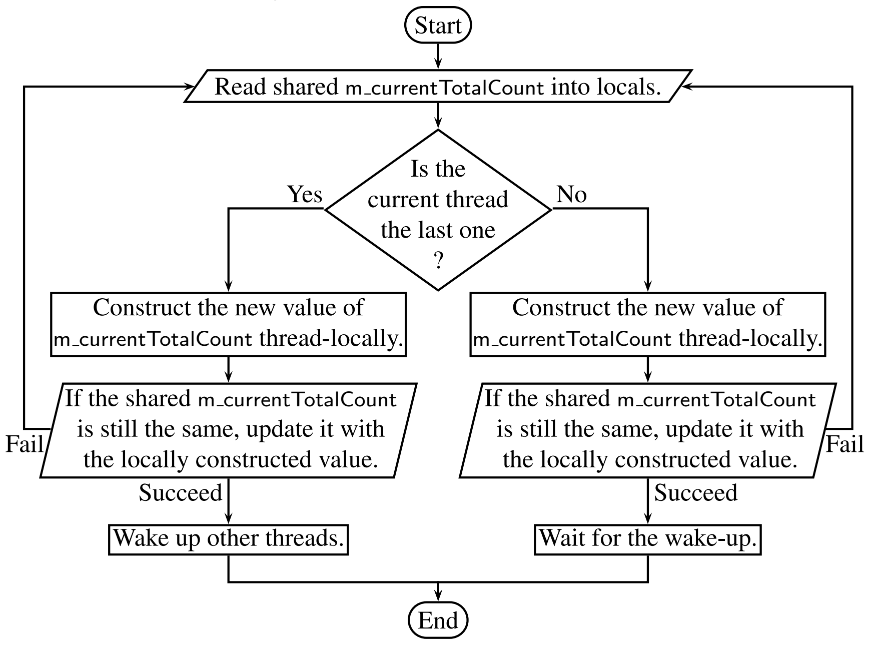
```
int count = 0; // the shared variable that will be incremented.
Barrier barrier = new Barrier(3); // create a barrier with three participants.
barrier.AddParticipants(2); // changed my mind, make it 5 participants.
barrier.RemoveParticipant(); // let us settle on 4 participants.
Action action = () => { // Defining the threads:
  Interlocked.Increment(ref count); // equivalent to an atomic count++
  barrier.SignalAndWait(); // when all threads arrive here, count=4.
  Interlocked.Increment(ref count);
  barrier.SignalAndWait(); // when all threads arrive here, count=8.
  Interlocked.Increment(ref count);
  barrier.SignalAndWait(); // when all threads arrive here, count=12.
  Interlocked.Increment(ref count);
  barrier.SignalAndWait(); // when all threads arrive here, count=16.
  assert(count==16);
};
Parallel.Invoke(action, action, action, action); // launch 4 participants.
barrier.Dispose(); // free the resources.
```

The barrier implementation is a variation of the central barrier. The implementation is 1741 lines long; it uses a 32-bit variable m_currentTotalCount to store in its different bits the following quantities:

- the lowest 15 bits are for the total number of participants,
- bit 15 is dummy,
- bits 16–30 store the number of participants that have already reached the barrier,
- bit 31 stores the sense flag that saves the parity of the number of episodes.

The variable m_currentPhase stores number of episodes; the sense flag of m_currentTotalCount copies the lowest bit of m_currentPhase. In each episode, m_currentPhase gets incremented and m_currentTotalCount gets updated.

Since the update of m_currentTotalCount is a complicated operation, it cannot be realized atomically: first, m_currentTotalCount is read into thread-private variables, then a newer value is constructed, then m_currentTotalCount is updated by a CAS (compare-and-swap, explained below in the code). A CAS atomically tests whether m_currentTotalCount still has the same value, and, if so, m_currentTotalCount is set to a constructed value. Otherwise CAS fails: another thread was faster and changed the value of m_currentTotalCount after the current thread had read m_currentTotalCount but before the current thread could attempt changing the value of m_currentTotalCount. In the failure case the process repeats: the variable m_currentTotalCount is read again. Schematically SignalAndWait executes as follows:


Start
Read shared m_currentTotalCount into locals.
Is the current thread the last one ?
Yes
No
Construct the new value of m_currentTotalCount thread-locally.
Construct the new value of m_currentTotalCount thread-locally.
If the shared m_currentTotalCount is still the same, update it with the locally constructed value.
If the shared m_currentTotalCount is still the same, update it with the locally constructed value.
Fail
Fail
Succeed
Succeed
Wake up other threads.
Wait for the wake-up.
End

The wake-up is realized with two instances of the .NET class ManualResetEventSlim: in an even (resp. odd) episode all the threads except the last one wait for the raising of m_evenEvent (resp. m_oddEvent). The last arriving thread reinitializes the opposite, i.e. odd (resp. even), event and raises the even (resp. odd) event.

The algorithm behind the .NET implementation of SignalAndWait() is displayed below:

```
assert(!m_disposed); // check whether the deallocator has been called.
bool sense;
int total, current, currentTotal;
while(true) {
  currentTotal = m_currentTotalCount; // atomically read the shared variable into a local one.
  GetCurrentTotal(currentTotal, out current, out total, out sense); // extract bits of this number.
  // current is the number of participants that have arrived at the barrier, total is the total number of
  // participating threads, sense is true iff the episode is even.
  // If total is 0, then the barrier was not meant to be used for any thread.
  assert(total); // but the current thread tries to participate, which is an error.
  // If sense≠ (m_currentPhase%2 == 0), another thread has already arrived at the barrier and updated
  // m_currentTotalCount, but has not yet signaled wake-up. If current=0, the current thread would start
  // using the barrier on which everyone else is waiting. Depending on the scheduling of the current thread
  // and the other thread, the current thread would later wait or continue (we won't list such schedules here).
  // Such a behavior were nondeterministic.
  assert(current || sense == (m_currentPhase % 2 == 0)); // forbid nondeterministic behavior.
  if(current+1 == total) { // this thread is the last one to arrive.
    // Try to prepare m_currentTotalCount for the next episode, internally realized via CAS:
    if(SetCurrentTotal(currentTotal, 0, total, !sense)) {
      // No thread has changed m_currentTotalCount since it was read above,
      // right now it has been successfully reinitialized.
      m_currentPhase++; // the next episode number is set.
      if(sense) { // the terminating episode is even.
        // Nobody listens to the odd event now, in the next episode all participants will wait for the odd event:
        m_oddEvent.Reset(); // prepare the odd event for waiting.
        m_evenEvent.Set(); // Wake up the current participants which are waiting for this even event.
      } else { // the terminating episode is odd.
        // Nobody listens to the even event now,
        // in the next episode all participants will wait for the even event:
        m_evenEvent.Reset(); // prepare the even event for waiting.
        m_oddEvent.Set(); // Wake up the current participants which are waiting for this odd event.
      } // end of if(sense)-else-endif.
      // The last thread has reinitialized the barrier and woken everyone else up.
      return; // This thread is sure that the other threads have arrived at their barriers.
    } // SetCurrentTotal(...) has failed, meaning that in the meantime some other thread
    // has changed m_currentTotalCount. Re-read it and restart the whole computation.
  } else { // current+1≠total, i.e., current+1 < total.
    // More threads will arrive or are arriving at the barrier. Tell them that this thread has arrived by
    // incrementing current via CAS inside SetCurrentTotal(). If no other thread has arrived since this thread
    // read m_currentTotalCount, the increment is successful and this thread should start waiting:
    if(SetCurrentTotal(currentTotal, current+1, total, sense)) break;
    // Otherwise some other thread has arrived since this thread read m_currentTotalCount, so this thread
    // should re-read it and restart the computation.
  } // end of if(current+1 == total)-else-endif.
  // Some other thread has intervened. Try to arrive at the barrier again.
} // end of the loop body.
// Wake-up phase. In even episodes wait for the evenEvent, in odd episodes wait for the oddEvent:
ManualResetEventSlim eventToWaitOn = sense ? m_evenEvent : m_oddEvent;
eventToWaitOn.Wait(); // wait for the signal from the last thread.
```

A small aside on m_currentPhase should be made. It is the only variable that can overflow in all the barrier algorithms we are looking at. The overflow does not affect our shortened representation of the .NET code as long as a machine with 2's complement representation of integers is used, since our representation checks only the parity of the variable.

II. Required reasoning.
We check
- that when a thread is about to terminate, count = 16,
- several assertions internal to the barrier implementation, e.g. that no use-after-dispose befalls.

The barrier implementation is finite-state due to the finite bit width of the all the variables. By code inspection we found that the executions of the client use only a small portion of that bit width (since only four threads are executed), so exhaustive state enumeration should succeed in proving the above properties. Certain calls to the libraries have to be modeled. For instance, a Boolean suffices to model an instance of the class ManualResetEventSlim.

III. Inductive invariant.
There is no need for a user to spell out the inductive invariant, as exhaustive state enumeration actually is a form of invariant generation.

IV. Encoded program.
We encoded the program in Promela, which is the input language of the Spin [22] bug finder. Though Spin does not support sequential procedures, Spin is great for large chunks of sequential code. So we inlined the sequentially called Barrier, AddParticipants, RemoveParticipants, Increment, SignalAndWait and Dispose. The invoked action directly corresponds to a concurrent procedure of which four copies are started.

V. Using the tool.
The standard method of storing states is insufficient: the number of reachable states exceeds the available space. This is why Spin optionally employs a compression technique that reduces space consumption at the cost of increased runtime, namely a set of states is viewed as a set of strings accepted by a regular automaton [23]. This technique reduces space consumption of many practically interesting programs. Using automata compression, Spin was able to verify the barrier with its client in 5720 s on an Intel® Core™ i5-520M CPU clocked at 2.4 GHz with 8 GB RAM clocked at 1066 MHz.

VI. Beyond the frontier.
During modeling we needed to provide code of some library functions and abstract away features which are not directly representable in Spin (e.g. lambda closures or exceptions). Reducing the dependency on manual rewriting is the next challenge, which requires modeling huge parts of the runtime environment, including libraries and the operating system.

8 A software barrier for weak memory

In this section we verify a software barrier algorithm for the weak memory model TSO (total store order) in `Checkfence`. We describe verification in the following steps: the algorithm, required reasoning, high-level proof structure, encoding of the proof and using the tool.

I. Algorithm.
The algorithm is an adaptation of the one from Section 4.1. It is a one-time central barrier. In a run of the barrier function, each thread decrements a shared counter and waits until it gets zero. Each of n threads has a unique identifier tid between 0 and $n-1$.

The algorithm is given below:

```
shared int* count; // *count initially stores the number of threads.
// During a run (*count) = the number of threads that have not decremented yet.
shared bool started[n] = {false,...,false}; // n auxiliary variables, used only to specify the barrier property.
// Each thread calls the following function:
void barrier() {
  started[tid] = true; // the current thread has started.
  // Memory accesses inside the atomic portion are not interrupted by memory accesses of other threads.
  atomic { // Specifying an atomic decrement.
    x = *count; // read the shared counter into a local register.
    x = x−1; // decrement the register.
    *count = x; // write the decremented value back into memory.
  }
  do { // await loop
    x = *count; // read the shared counter into a local register.
    spinforzero = (x≠0); // spinforzero is true iff x is nonzero.
  } while(spinforzero); // wait until the read value is zero.
  assert(∀ threads t ≠ tid: started[tid]); // check the barrier property.
}
```

The program which we analyze is the initialization code followed by the parallel composition of the function bodies.

II. Required reasoning.
Not surprisingly, we will not mention invariants here, since the concept of a transition system is absent in the definition of most weak memory models. Instead, the semantics of TSO is given axiomatically, restricting the executions of the whole system and the executions of each thread. The barrier loops are soundly unrolled and the axioms are applied to length-bounded executions. The existence of a certain execution is encoded as a formula; after conversions into the Boolean form, applying a decision procedure for SAT suffices.

To describe sound loop unrolling, notice first that all iterations of each thread's loop except the last one are side-effect free: they change neither the shared state, nor the termination condition, which is spinforzero here, nor the later used local state. Thus dropping any number of loop iterations produces an equivalent execution as long as at least one iteration remains. Then it suffices to distinguish two kinds of executions for each thread.

- In the first kind, the loop of the thread is executed finitely often. The tool considers an equivalent execution in which all except the last iterations are elided, i.e. the loop is replaced with its body followed by assuming the termination condition $\neg$spinforzero.
- In the second kind, the loop diverges. The tool considers an equivalent execution in which the loop body executes exactly once after all the stores have happened.

III. High-level proof structure.
We now give a high-level description of the ingredients of the formula that encodes the existence of an error trace.

The TSO memory model imposes a total order on loads, a total order on stores, as well as a certain ordering of atomic load-and-stores executed by different processors. The semantics makes these notions precise in terms of the global traces of the whole multithreaded program and the traces of separate threads. The exact semantics is described in [9], while we are going to pick up the interesting highlights informally.

A *local trace* of a thread consists just of

- the instructions of the executing processor,
- the addresses and the values that these instructions use,
- the control flow order between these instructions,
- the information about what instructions belong to the same atomic load-store,
- the orderings resulting from data-to-data and data-to-control dependencies between instructions,

A *global trace* consists of the following components:

- the instructions of all the processors, equipped by the thread identifiers,
- the used addresses and values,
- the control flow order between the instructions,
- the mapping of loads to stores that provide the value,
- the information about what subinstructions form a part of the same atomic block,
- the orderings resulting from data-to-data and data-to-control dependencies between instructions,
- the execution status (is repeating a loop, is throwing an exception or is normally continuing).

For verification, executions of the bounded version are checked against the user-given assertions. The existence of an execution violating a user-given specification is equivalent to the satisfiability of a particular formula

$$\Psi \wedge \Theta \wedge \bigwedge_{k<n} \Pi_k \wedge S, \tag{1}$$

where Ψ constrains on the global trace, Θ constrains the memory model, Π_k constrain the local traces ($k < n$) and S is the error condition. We are going to describe the structure of the above formula next.

IV. Encoding of the proof.
This subsection provides major low-level details of the formula (1).

The free variables of the aforementioned formula are

- data variables representing the values used in a load or in a store instruction, one variable per instruction,
- address variables used in a load or in a store instruction, one variable per instruction,
- Booleans marking whether an instruction is really executed or not (the code might contain conditionals), one variable per instruction,
- the execution status with which a thread terminates, one variable per thread,
- Booleans reporting whether a load sees the value provided by a store, one variable per each load-store pair,
- Booleans marking whether two instructions are ordered by the memory order, one variable per pair of different instructions.

The conjunct Ψ stipulates that the above variables actually encode a global trace. For example, whenever a load sees the value provided by a store, both instructions should be executed and mention the same address and the same value.

The conjunct Θ restricts the global trace by the constraints of the memory model. One such fact is, for instance, that the memory order is total on stores and total on loads.

The conjunct Π_k restricts the local trace by local constraints, e.g. on the control flow order of thread k ($k < n$). The values and addresses of the instructions, as well as the conditions under which the threads terminate without error or loop bound overrun, are obtained in two steps. First, the symbolic evaluation of the unrolled thread provides the terms (i.e. mathematical functions), whose application to the loaded values would give the final values, addresses and Booleans. Second, these functions are used verbatim to construct the formula Π_k ($k < n$). Using the fact that the values, the addresses and the terms are bounded, one can transform such Π_k into pure Boolean formulas ($k < n$).

Finally, the quantifiers are replaced by conjunctions and disjunctions using the fact that the trace is bounded.

V. Using the tool.
The tool `Checkfence`, written in C++ under Windows, was no longer compilable when we started our research; we have revived it and ported it to Linux. Verifying the assertion for the number of the threads varying between 1 and 8 took the following time on an Intel® Core™ i5-520M CPU clocked at 2.4 GHz with 8 GB RAM clocked at 1066 MHz:

Threads	1	2	3	4	5	6	7	8
Time, s	0	0.01	0.03	0.11	0.32	2.06	11.19	189.45

VI. Beyond the frontier.
The `Checkfence` tool limits verification to 8 threads. At the time this research was conducted, for weak memory there were no other tools for barrier verification that were suitable (i.e. obtainable, usable and supported). Doubtless the `Checkfence` approach gives more confidence in correctness than testing or manual code inspection; but it would be useful to verify the algorithm for an unbounded number of threads. This next challenge is left for future research.

9 Discussion

This section discusses the significance of the barriers in general, supports our choice of the verification approaches, mentions the related work and presents future work.

The importance of software barriers. The software barrier primitive is indispensable in parallel programming. In parallel scientific computing, barriers are frequently used, for example, in Gaussian elimination [13], matrix multiplication [7], random number generation [10], LU factorization [4], Jacobi method [14], fluid simulation [5] and solving Navier-Stokes equations [51], protein simulation and ray tracing [8], ocean circulation models [24] and in many other applications. Barriers are heavily used in high-performance computing. Today, the software barrier is a standard synchronization primitive in PThreads, OpenMP, CUDA (all described in [17]), java.util.concurrent [16], .NET [42] and MPI [35].

In the formal methods community, barriers are also used, e.g. to model synchronous execution in interleaving semantics [6] or as a part of an execution model [21].

There is significant variation in algorithms that implement software barriers. Herlihy and Shavit's textbook [20] describes the central barrier (which is simple but slow), the combining tree barrier and the static tree barrier (which are faster but trickier to implement). The dissemination barrier (which does not have a wake-up phase) and the tournament barrier (which is the fastest on many shared-memory architectures) are given as exercises to be worked out. The combining tree barrier uses a more general concept of the software combining tree [50]; numerous variations thereof have been studied [31, 38]. The dissemination barrier is an instance of a more general parallel prefix computation; the tournament barrier has an exceptionally low communication overhead [19]. Simple counting barriers are easily extendible with features; e.g. the barrier of .NET [39, 40] allows adding and removing participating threads, wait cancellation, timeouts and is fuzzy: it permits a thread to execute useful work while waiting for the other threads. Faster fuzzy barriers are tree-based [18,45].

Choice of the verification approach. A reader might wonder why we are not using other verifiers. Indeed, we had acquaintance with more verifiers but found that Boogie, Checkfence, Jahob, VCC and SPIN are best suited for our experiments.

In the formal methods community it is generally believed that there is no single simple verification approach that suits any verification task. We have demonstrated that even if we wish to analyze a set of programs that all serve the same purpose—barrier synchronization here—a juxtaposition of several reasoning techniques might be required to handle them all. Thus our experience confirms the expectation of the community.

For instance, we found out that in certain cases Jahob suits best while in others Boogie is a better match. Both tools are similar in that they require the user to provide the inductive invariant, which is then translated into verification conditions to be discharged by theorem provers. The tools are different in that they place their emphases on supporting different logics: one should always use a tool with appropriate decision support for the logic required for the inductiveness proof.

Related work. So far verification of barriers has been sporadic and unsystematic, usually in the context of other projects or non-automatic [6,12,25,26,34,47]. Most barrier algorithms have not been machine-verified at all to the best of our knowledge. The closest sequential counterpart for the logarithmic barriers that has been verified is the composite pattern [2], in which there is a bottom-up tree traversal from a node to a root. The composite pattern is a challenge to AUFLIA solvers [30], which cannot reason about trees natively, and even adding axioms describing the theory of trees requires nontrivial hacking of triggers.

Hobor and Gherghina [21] add a barrier statement to concurrent separation logic; however, no program verification takes place. Aiken and Gay [1] verify the synchronization patterns of programs with barriers. All such results assume correctness of the barrier implementations. In contrast, we actually verify the implementations.

Future work. Throughout the paper we have provided several challenging verification tasks that still lie beyond the automatic verification frontier. Based on our experiments, we expect that more elaborate barrier algorithms (e.g. fuzzy and adaptive barriers [18,45]) cannot be verified automatically now, but are amenable to interactive verification and bug finding. By plain inspection, we found a minor bug in the high-level description of such a barrier [45], which, however, was repaired in the low-level code [36].

It would be also interesting to connect the verification of barriers with the verification of clients via some interface, e.g. by using a variant of rely-guarantee reasoning or other logics for concurrency.

Currently there is significant variation in the reasoning principles behind the correctness proofs of the barriers. We are unaware of any verification tool that provides enough support for all of them. However, it would be interesting and useful to have a verification system with which all the barriers could be verified.

Still another line of future work touches liveness properties of barriers. The following property, currently outside the scope of automatic verification, is useful: once all threads have started, eventually all threads will successfully terminate.

Conclusion. We have considered several software barrier algorithms, an implementation in C, a barrier on weak memory and a barrier with its client. Most of these benchmarks have previously been on the other side of the verifiability frontier, not amenable to automatic methods. We have verified these benchmarks as automatically as the state of the art permits. On one hand, verification has increased our certainty in the correctness of those benchmarks by orders of magnitude. On the other hand, we have improved the Jahob tool, pushing the verification frontiers even further. Our experiments are publicly available, reproducible and documented. We have identified more complicated versions of the barriers that represent future challenges.

Acknowledgments This work would be virtually impossible without valuable help of many researchers. Jasmin Blanchette helped us improving the typesetting and the grammar of the paper. Sebastian Burckhardt supplied us with the Checkfence tool and assisted us in creating usable inputs. Ernie Cohen showed the main ingredients of the auxiliary variables that were needed to get rid of reasoning about cardinality constraints in VCC. Viktor Kuncak provided us with insights on the internals of Jahob. Rustan Leino taught us the ultimate way of debugging Boogie specifications. Michał Moskal described a way to express a global invariant using VCC annotations and also fixed VCC bugs that we discovered. Leonardo de Moura produced the usable versions of the SMT prover Z3, fixing the bugs that we have discovered in Z3. Peter Müller tried to repeat the static tree barrier experiments in Dafny and drew our attention to supporting verification by axioms. Sriram Rajamani gave valuable comments on an earlier draft. Thomas Wies provided comments on the internals of Jahob and rescued us while improving and bugfixing Jahob. Finally, we thank the anonymous reviewers for their helpful comments.

The work has been conducted at the IMDEA Software Institute and partially supported by Madrid Regional Government MINECO Project TIN2009-14599-C03-02 Desafios, EU NoE Project 256980 Nessos and Amarout Europe Project 291803. We acknowledge additional financial support of the German Academic Exchange Service and infrastructural support of the Chair for Foundations of Software Reliability and Theoretical Computer Science at Technische Universität München.

References

1. Alexander Aiken and David Gay. Barrier inference. In David Bruce MacQueen and Luca Cardelli, editors, *ACM Symposium on Principles of Programming Languages*, pages 342–354. ACM, 1998.
2. Jonathan Aldrich, Mike Barnett, Dimitra Giannakopoulou, Gary Todd Leavens, and Natasha Sharygina, editors. *SAVCBS'08 workshop at SIGSOFT 2008/FSE 16, November 9-10, 2008, proceedings, Technical Report CS-TR-08-07*, 2008.
3. Abdelwaheb Ayari. *System verification tools based on Monadic Logics*. PhD thesis, University of Freiburg, 2003.
4. Muhammad S. Benten and Harry F. Jordan. Multiprogramming and the performance of parallel programs. In Rodrigue [44], pages 374–383.
5. Christian Bienia. PARSEC—the Princeton application repository for shared memory computers, August 2009. `http://parsec.cs.princeton.edu`, version 2.1, retrieved on 5 January 2011.
6. Peter Braun, Heiko Lötzbeyer, and Oscar Slotosch. Quest users guide. Technical report, Technische Universität München, March 2000.
7. Eugene D. Brooks III, Timothy S. Axelrod, and Gregory A. Darmohray. The Cerberus multiprocessor simulator. In Rodrigue [44], pages 384–390.
8. J. Mark Bull, Robert A. Davey, Robin Freeman, P. J. Graham, David S. Henty, Mark E. Kambites, Jan Obdrzálek, L. Pottage, Lorna A. Smith, S. D. Telford, and Martin D. Westhead. The Java Grande benchmark suite, 2001. `http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html`, accessed on 5 June 2013.
9. Sebastian Burckhardt. *Memory Model Sensitive Analysis of Concurrent Data Types*. PhD thesis, University of Pennsylvania, 2007.
10. William Celmaster. Implementation of the acceptance-rejection method on parallel processors: A case study in scheduling. In Rodrigue [44], pages 131–136.
11. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC—the Verifying C Compiler, 2012. `http://vcc.codeplex.com`, accessed on 7 June 2013.

12. Joseph Cordina, Stephen Fenech, and Gordon J. Pace. Model checking concurrent assembly algorithms. Technical report, Departments of Computer Science and AI, University of Malta, 2007.
13. Gregory A. Darmohray and Eugene D. Brooks III. Gaussian techniques on shared memory multiprocessor computers. In Rodrigue [44], pages 20–26.
14. John Emory Dennis Jr., José Mario Martínez, and Xiaodong Zhang. Parallel block triangular decompositions for solving sparse nonlinear systems of equations. In Jack Dongarra, Ken Kennedy, Paul Messina, Danny Chris Sorensen, and Robert Gary Voigt, editors, *PPSC*, pages 168–173. SIAM, 1991.
15. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In Zhong Shao and Benjamin Crawford Pierce, editors, *ACM Symposium on Principles of Programming Languages*, pages 2–15. ACM, 2009.
16. Jeff Friesen. *Beginning Java 7*. Apress, November 2011. ISBN 978-1-4302-3909-3.
17. Fayez Gebali. *Algorithms and parallel computing*. John Wiley & Sons, Inc., March 2011. ISBN 978-0-470-90210-3.
18. Rajiv Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In Joel S. Emer, editor, *Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63. ACM Press, 1989.
19. Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17:1–17, February 1988.
20. Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
21. Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic. In Gilles Barthe, editor, *Programming Languages and Systems, European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 276–296. Springer, 2011.
22. Gerard Johan Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. `http://www.spinroot.com`, accessed on 7 June 2013.
23. Gerard Johan Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *Intl. Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.
24. Hsiao-Ming Hsu, Jih-Kwon Peir, and Dale B. Haidvogel. Performance of an ocean circulation model on LCAP. In Rodrigue [44], page 285.
25. Thuan Quang Huynh and Abhik Roychoudhury. A memory model sensitive checker for C#. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 476–491. Springer, 2006.
26. Bart Jacobs. Verified general barriers implementation, 2010. `http://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/barrier.c.html`, retrieved on 7 February 2013.
27. Viktor Kuncak, Thomas Wies, Karen Zee, Alexander Malkis, Charles Bouillaguet, Huu Hai Nguyen, and Peter Schmitt. Jahob verification system. The tool site is at `http://lara.epfl.ch/w/jahob_system`, the improved source code is at `http://www.sec.in.tum.de/~malkis/jahob.7z`, accessed on 9 March 2016.
28. Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In Ana Cavalcanti and Dennis Dams, editors, *Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809. Springer, 2009.
29. K. Rustan M. Leino. This is Boogie 2. Technical Report KRML 178, Microsoft Research, June 2008.
30. K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In *TOOLS & EXPERIMENTS Workshop*, 2010.
31. Boris Dmitrievich Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. *International Journal of Parallel Programming*, 19(3):225–250, 1990.
32. Alexander Malkis and Anindya Banerjee. Detailed input and comments on the verification tools applied to software barriers, 2011. Available at `http://www.sec.in.tum.de/~malkis/BarrierVerification` and `http://software.imdea.org/~ab/BarrierVerification`, accessed on 9 March 2016.
33. Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer, 1995.
34. Olga Shumsky Matlin, Ewing L. Lusk, and William McCune. SPINning parallel systems software. In Dragan Bosnacki and Stefan Leue, editors, *SPIN*, volume 2318 of *Lecture Notes in Computer Science*, pages 213–220. Springer, 2002.
35. John M. May. *Parallel I/O for high-performace computing*. Academic Press, 2001. ISBN 1-55860-664-5.
36. John Michael Mellor-Crummey and Michael Lee Scott. Barriers for the BBN Butterfly 1. `ftp://ftp.cs.rochester.edu/pub/packages/scalable_synch/locks_and_barriers/Bfly1.tar.Z`, retrieved on 16 February 2013.
37. John Michael Mellor-Crummey and Michael Lee Scott. Barriers for the Sequent Symmetry. `ftp://ftp.cs.rochester.edu/pub/packages/scalable_synch/locks_and_barriers/Symmetry.tar.Z`, retrieved on 16 February 2013.

38. John Michael Mellor-Crummey and Michael Lee Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.
39. Microsoft Corp. .NET framework libraries, 2008. `http://referencesource.microsoft.com/netframework.aspx`, version 4, file Barrier.cs, retrieved on 23 May 2011.
40. Microsoft Corp. MSDN barrier documentation, 2011. `http://msdn.microsoft.com/en-us/library/system.threading.barrier.aspx`, sample C# code, retrieved on 5 July 2011.
41. Michał Moskal, Wolfram Schulte, Ernie Cohen, Mark A. Hillebrand, and Stephan Tobies. Verifying C programs: A VCC tutorial, 2012. Retrieved from `http://www.codeplex.com/Download?ProjectName=VCC&DownloadId=476507` on 23 July 2011.
42. Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, and Morgan Skinner. *Professional C# 2012 and .NET 4.5*. John Wiley & Sons, Inc., October 2012. ISBN 978-1-1183-1442-5.
43. Virgile Prevosto and Uwe Waldmann. SPASS+T. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *ESCoR: FLoC'06 Workshop on Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 18–33, Seattle, WA, USA, 2006.
44. Garry Hector Rodrigue, editor. *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing, Los Angeles, California, USA, December 1-4, 1987*. SIAM, 1989.
45. Michael Lee Scott and John Michael Mellor-Crummey. Fast, contention-free combining tree barriers for shared-memory multiprocessors. *Int. J. Parallel Program.*, 22:449–481, August 1994.
46. Michael Lee Scott and John Michael Mellor-Crummey. Pseudocode of scalable synchronization, 1994. `http://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html`, retrieved on 23 February 2013.
47. Anton Smit. Verifying a barrier algorithm with a mechanical theorem prover, 2001. Master thesis, Faculty of Mathematics and Natural Sciences, University of Groningen.
48. Philippe Suter, Robin Steiger, and Viktor Kuncak. Sets with cardinality constraints in satisfiability modulo theories. In Ranjit Jhala and David A. Schmidt, editors, *Intl. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2011.
49. Thomas Wies, Ruzica Piskac, and Viktor Kuncak. Combining theories with shared set operations. In Silvio Ghilardi and Roberto Sebastiani, editors, *Frontiers of Combining Systems*, volume 5749 of *Lecture Notes in Computer Science*, pages 366–382. Springer, 2009.
50. Pen-Chung Yew, Nian-Feng Tzeng, and Duncan Hamish Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Computers*, 36(4):388–395, 1987.
51. Sam Yu and A. Daniel Kowalski. A study of parallel numerical algorithms for the solution of the Navier-Stokes equation. In Jack Dongarra, Paul Messina, Danny Chris Sorensen, and Robert Gary Voigt, editors, *PPSC*, pages 285–290. SIAM, 1989.