A version of this technical report has been accepted for publication at 15th International Conference on Formal Engineering Methods (ICFEM 2013).

Verification of Static and Dynamic Barrier Synchronization Using Bounded Permissions

Duy-Khanh Le, Wei-Ngan Chin, Yong-Meng Teo

Department of Computer Science, National University of Singapore
{leduykha,chinwn,teoym}@comp.nus.edu.sg

(Technical Report - August 2013)

Abstract. Mainstream languages such as C/C++ (with Pthreads), Java, and .NET provide programmers with both *static* and *dynamic barriers* for synchronizing concurrent threads in fork/join programs. However, such barrier synchronization in fork/join programs is hard to verify since programmers must not only keep track of the dynamic number of participating threads, but also ensure that all participants proceed in correctly synchronized phases. As barriers are commonly used in practice, verifying correct synchronization of barriers can provide compilers and analysers with important phasing information for improving the precision of their analyses and optimizations.

In this paper, we propose an approach for statically verifying correct synchronization of *static* and *dynamic barriers* in fork/join programs. We introduce the notions of *bounded permissions* and *phase numbers* for keeping track of the number of participating threads and barrier phases respectively. The approach has been proven sound, and a prototype of it (named VeriBSync) has been implemented for verifying barrier synchronization of realistic programs in the SPLASH-2 benchmark suite.

1 Introduction

Software barriers are a kind of collective operations available in Pthreads, Java, .NET, OpenMP, and others. Threads participating in a barrier proceed in *phases*. A typical usage of barriers is presented in Fig. 1. When a thread issues a barrier wait, it waits until a pre-defined number of threads (all threads or just a group of threads) have also issued a barrier wait; after that, all participating threads proceed to the next phase. SPMD (single program, multiple data) programs, such as those written in OpenMP, typically have a single barrier to coordinate all threads in the programs. On the other hand, fork/join programs written in Pthreads, Java, and .NET could use more than one barrier to coordinate different (possibly non-disjoint) groups of threads. In Pthreads [1], barriers are *static*, i.e. the number of participants is fixed. In .NET framework [7], barriers are *dynamic* as the number of participants can vary during a program's execution. The java.util.concurrent library [9] supports both static and dynamic barriers (i.e. CyclicBarrier and Phaser respectively).

```
 //b has 2 participants
  b = new barrier(2);

//Thread 1 || //Thread 2
//Phase 0  || //Phase 0
wait(b);   || wait(b);
//Phase 1  || //Phase 1
```

Fig. 1. Typical Usage of Barriers

Barriers are commonly used in practice. For example, all twelve realistic programs in SPLASH-2 benchmark suite [25] use at least one barrier and four out of twelve programs use more than one barrier for synchronization, covering numerous application domains such as computer graphics (volrend), water molecule simulation (water-spatial), and engineering (radix) among others. Therefore, verifying correct synchronization of barriers is desirable because it can provide compilers and analysers with important phasing information for improving the precision of their analyses and optimizations such as reducing false sharing [12], may-happen-in-parallel analysis [16, 27], and data race detection [14]. For example, given the information that a program is verified as correctly synchronized on a barrier, concurrency analysers [14, 16, 27] could significantly improve their analyses by exploiting the fact that two statements in different barrier phases cannot be executed in parallel. However, static verification of barrier synchronization in fork/join programs is hard because programmers must not only keep track of (possibly dynamic) number of participating threads, but also ensure that all participants proceed in correctly synchronized phases.

Verification approaches such as those based on separation logic [21] and implicit dynamic frames [24] often use an *access permission system*, such as fractional permissions [5] or counting permissions [4], as the basis for reasoning about race-free sharing of resources. There are *bounded resources* (e.g. barriers) which are typically shared among a *bounded* number (or a group) of concurrent threads. Unfortunately, when using existing permission systems [4, 5], a resource could be split off an unbounded number of times and hence unintentionally shared among an *unbounded* number of concurrent threads. Therefore, existing permission systems are not suitable for reasoning about bounded resources.

In this paper, we first introduce a new permission system, called *bounded permissions*, to enable reasoning for bounded resources (§3). We then present a logical approach for statically verifying correct synchronization of *static* and *dynamic* barriers in fork/join programs. For verifying *static barriers*, the approach uses *bounded permissions* and *phase numbers* to keep track of the number of participants and barrier phases respectively (§4). For verifying *dynamic barriers*, the approach introduces *dynamic bounded permissions* to additionally keep track of the additions and/or removals of participants (§5). To the best of our knowledge, our paper is the first effort to verify synchronization of both *static* and *dynamic barriers* in fork/join programs.

2 Background

In this section, we first discuss some basic notations in separation logic [20, 21]. We then present a fork/join programming language with barriers.

2.1 Concurrent Separation Logic and Permissions

Separation logic [21] is a resource logic for reasoning about heap-manipulating programs. In separation logic, the simplest *heaps* are the *empty heap* `emp` and the *heap node* $x \mapsto E$. The basic heap node $x \mapsto E$, pronounced *x points to E*, asserts that it consists of a single cell with integer address x and integer content E. We write $x \mapsto \_$ to describe a heap node with unknown content. Heaps are connected together to form larger heaps by using the *separation connective* $*$. In order to reason about race-free sharing of resources among concurrent threads, heaps are enhanced with *permissions* π [4, 5]. A heap node $x \overset{\pi}{\mapsto} E$ indicates a permission to access the content E at the address x. A permission can be *partial* or *full* indicating read or write permission respectively. A permission (either full or partial) can be split into multiple partial permissions which can be shared among threads. Partial permissions can also be gathered back into a single full permission for accounting. A *memory state* consists of $*$-conjunctions of heaps and constraints on their addresses, contents, and permissions.

The beauty of separation logic lies under its *frame rule*:

$$\frac{\{\Phi_1\}\ \mathtt{s}\ \{\Phi_1^{'}\} \qquad modifies(\mathtt{s}) \cap FV(\Phi_2){=}\varnothing}{\{\Phi_1 * \Phi_2\}\ \mathtt{s}\ \{\Phi_1^{'} * \Phi_2\}} \tag{1}$$

Informally, if a statement `s` is safe in a state Φ_1, then `s` is also safe in a larger state $\Phi_1 * \Phi_2$ given the side condition that `s` does not modify any free variables in Φ_2 [21]. The same principle applies when verifying concurrent threads, as indicated in the following *parallel composition rule*:

$$\frac{\begin{array}{ll}\{\Phi_1\}\ s_1\ \{\Phi_1^{'}\} & modifies(s_1) \cap FV(\Phi_2, \Phi_2^{'}){=}\varnothing \\ \{\Phi_2\}\ s_2\ \{\Phi_2^{'}\} & modifies(s_2) \cap FV(\Phi_1, \Phi_1^{'}){=}\varnothing\end{array}}{\{\Phi_1 * \Phi_2\}\ s_1 \mathtt{||} s_2\ \{\Phi_1^{'} * \Phi_2^{'}\}} \tag{2}$$

Since two concurrent threads s_1 and s_2 "mind their own business" and do not modify variables of each other, the combined state $\Phi_1^{'} * \Phi_2^{'}$ is safe [20]. This principle allows *local reasoning* as concurrent threads are verified independently. Note that the frame rule can be expressed in terms of the parallel composition rule as `s` is equivalent to `s||no-op`.

2.2 A Fork/Join Programming Language with Barriers

Mainstream languages such as C/C++ (with Pthreads), Java, and .NET provide barriers for synchronizing a group of threads. As our approach is language-independent, we develop a core language with fork/join concurrency and barriers (Fig. 2). The language is straightforward (see Appendix A for details). Note that in this paper, for brevity of presentation, we often use the parallel composition ($s_1||s_2$); as an abbreviation for creating concurrent threads. The parallel composition is syntactic sugar and can easily be encoded via fork and join.

$P ::=$	$proc^*$	Program
$proc ::=$	$\mathtt{pn}((t\ v)^*)\ \ spec^*\ \{\ s\ \}$	Procedure declaration
$spec ::=$	**requires** Φ_{pr} **ensures** Φ_{po};	Pre/Post-conditions
$t ::=$	`int` | `bool` | `void` | `barrier`	Type
$e ::=$	$v \mid k \mid e_1{=}e_2 \mid e_1{\neq}e_2 \mid \ldots$	Expression
$s ::=$	$v =$ **fork**(pn,v^*) | **join**(v) | `barrier` $b =$ **new** `barrier`(n) | **destroy**(b) | **wait**(b) | **add**(b, m) | **remove**(b, m) | $s_1; s_2$ | `pn`(v^*) | **if** e **then** s_1 **else** s_2 | $\ldots$	Statement

Fig. 2. Fork/Join Programming Language with Specifications

3 Bounded Permissions

In this section, we present our bounded permission system for reasoning about bounded resources. Although we place our bounded permissions in the context of separation logic, bounded permissions can be generally applied to other logics such as implicit dynamic frames [24].

A permission system should distinguish full permission for total control (read, write, and destroy) from partial permission for shared access (read only: no thread can write or destroy) [5]. Permission accounting (e.g. the ability to split a permission into multiple partial permissions for shared access and to combine partial permissions into a full permission for exclusive write) is critical for reasoning about fork/join programs [4]. Besides the above properties, our bounded permission system additionally provides the notion of "boundedness" as the guarantee for reasoning about bounded resources.

Bounded permission: $x \xmapsto{c,t} E$

Permission count:	c	Full permission:	$c = t$
Permission total:	t	Partial permission:	$c < t$
Permission invariant:	$0 < c \leq t$	Unit permission:	$c = 1$

Permission rules:

[SPLIT/COMBINE] $x \xmapsto{c,t} E \wedge c{=}c_1{+}c_2 \iff x \xmapsto{c_1,t} E * x \xmapsto{c_2,t} E$

[SEP] $x_1 \xmapsto{c_1,t_1} E * x_2 \xmapsto{c_2,t_2} E \wedge (t_1{\neq}t_2 \vee c_1{+}c_2{>}t_1) \Longrightarrow x_1{\neq}x_2$

Fig. 3. Bounded Permission System

Fig. 3 summarizes our bounded permission system. An assertion $x \xmapsto{c,t} E$ represents a bounded permission to access the content E at the address x. A permission quantity is a pair of integers (c,t) where $0{<}c{\leq}t$; $c{=}t$ indicates a *full permission* while $c{<}t$ indicates a *partial permission*. Permissions with $c{=}1$ are called *unit permissions*. A permission can be split into two permissions (reading from left to right of the rule [SPLIT/COMBINE]). In the other direction, heap nodes can be combined using $*$ iff their addresses coincide, they agree on their contents and their permissions can be combined arithmetically. Note that due to the invariant $0{<}c{\leq}t$, a unit permission cannot be split off. Besides the ability to split/combine permissions, the notion of separation ([SEP]) is important for reasoning about separation of resources [4, 21]. Two heaps agreeing on their contents are separated $(x_1{\neq}x_2)$ if their permission totals are different or the sum of their permission counts is higher than the permission total.

We can create a new bounded-permission resource (with n is assigned to the permission total) and destroy it only in full permissions:

$$\begin{array}{c} \{\ n > 0\ \}\ \texttt{x = new(n);}\ \{\ x \xmapsto{n,n} \_\ \} \\ \{\ x \xmapsto{n,n} \_\ \}\ \texttt{destroy(x);}\ \{\ \texttt{emp}\ \} \end{array} \tag{3}$$

Given a full permission, we are sure that no other thread can access the shared resource. Therefore, we can safely destroy it. In languages with automatic garbage collection, such a destroy operation is not necessary, but the full permission is still useful in guiding the garbage collector for safe collection.

Similarly, we need a full permission for writing and any permission (full or partial) for reading:

$$\begin{array}{l} \{\ x \xmapsto{n,n} \_\ \}\ \texttt{[x] = E;}\ \{\ x \xmapsto{n,n} E\ \} \\ \{\ x \xmapsto{c,t} E\ \}\ \texttt{y = [x];}\ \{\ x \xmapsto{c,t} E \wedge y = E\ \} \end{array} \qquad (4)$$

`[x]` is an abbreviation for accessing the content located at the address `x`. In the last rule, there is a side condition that y is not free in E.

Now, it is straightforward to verify the correctness of the program in Fig. 4, in which only two threads are intended to concurrently read the content at the location `x`. As a brief comparison, when using existing permission systems [4, 5], there is nothing to prevent `x` from being split off into more than two partial permissions and hence unintentionally accessed by more than two threads.

$$\begin{array}{c}
\{\ \texttt{emp}\ \} \\
\texttt{x = new(2);} \\
\{\ x \xmapsto{2,2} \_\ \} \\
\texttt{[x] = 5;} \\
\{\ x \xmapsto{2,2} 5\ \} \\
//\underline{[\textsc{split}]} \\
\left(\begin{array}{l} \{\ x \xmapsto{1,2} 5\ \} \\ \texttt{y=[x]+1;} \\ \{\ x \xmapsto{1,2} 5 \wedge y = 6\ \} \end{array} \middle\| \begin{array}{l} \{\ x \xmapsto{1,2} 5\ \} \\ \texttt{z=[x]-1;} \\ \{\ x \xmapsto{1,2} 5\ \wedge z = 4\ \} \end{array} \middle\| \begin{array}{l} \{\ \texttt{emp}\ \} \\ \texttt{t=10;} \\ \{\ t = 10\ \} \end{array}\right)\texttt{;} \\
//\underline{[\textsc{combine}]} \\
\{\ x \xmapsto{2,2} 5 \wedge y = 6 \wedge z = 4 \wedge t = 10\ \} \\
\texttt{destroy(x);} \\
\{\ \texttt{emp} \wedge y = 6 \wedge z = 4 \wedge t = 10\ \}
\end{array}$$

Fig. 4. Example of Using Bounded Permissions

The following lemma states our guarantee on boundedness property.

Lemma 1 (Boundedness) *Given a resource x with a full permission $x \xmapsto{n,n} \_$ ($n>0$), there are at most n concurrent accesses to x, i.e. x is shared among at most n concurrent threads at a given time.*

Proof. A thread needs at least a unit permission $x \xmapsto{1,n} \_$ to access x and there are at most n such unit permissions. □

4 Verification of Static Barriers

In this section, we present our approach to verifying correct synchronization of static barriers. We first define what it means for a program to be correctly synchronized.

Definition 1 (Correct Synchronization) *A program is correctly synchronized with respect to a static barrier b iff:*

- *There is exactly a predefined number of threads participating in the barrier b's wait operations.*
- *Participating threads operate on b in the same numbers of phases.*

```
                        { emp }
              barrier b = new barrier(2);
            { b \xmapsto{2,2} barrier(0) }
{ b \xmapsto{1,2} barrier(0) } || { b \xmapsto{1,2} barrier(0) }
//phase 0;                     ||  //phase 0;
wait(b);                       || wait(b);
//phase 1;                     ||  //phase 1;
{ b \xmapsto{1,2} barrier(1) } || { b \xmapsto{1,2} barrier(1) }
            { b \xmapsto{2,2} barrier(1) }
                   destroy(b);
                        { emp }
```

(a) Correctly synchronized

```
                        { emp }
              barrier b = new barrier(2);
            { b \xmapsto{2,2} barrier(0) }
{ b \xmapsto{1,2} barrier(0) } || { b \xmapsto{1,2} barrier(0) }
//phase 0;                     || //phase 0;
wait(b);                       || //no-op;
//phase 1;                     ||
{ b \xmapsto{1,2} barrier(1) } || { b \xmapsto{1,2} barrier(0) }

                        //FAIL
                          ...
```

(b) Incorrectly synchronized

Fig. 5. Barrier Synchronization

For illustration, the program in Fig. 5a is correctly synchronized while the program in Fig. 5b is not because the two threads in Fig. 5b operate in different numbers of phases. As shown in Section 3, bounded permissions can be used to ensure that *at most* a predefined number of threads can access a resource at a given time. However, verification of barrier synchronization requires a stronger guarantee: *exactly* a predefined number of threads participates in a barrier wait. We enforce such a guarantee by requiring that a participating thread must hold a *unit permission* to perform a barrier wait. If a participant has more than a unit permission, it prohibits other participants from participating. An analogy is a meeting room with n keys distributed among n participants; a meeting takes place only when all participants have come. If a participant has more than one key, when he/she enters the room, at least one other participant will not be able to get in and the meeting cannot take place. We capture barrier phasing by using *phase numbers*, which increase by one after each barrier wait, and require that all participants end up with the same phase numbers. If participants have different phase numbers when completing their execution, some of them must have lost phasing and the program is not correctly synchronized.

A summary of our approach is presented in Fig. 6. An assertion $b \xmapsto{c,t} \mathsf{barrier}(p)$ indicates a bounded permission (c, t) to access the barrier b which is at phase p. When creating a new barrier with the number of participants n, a full permission (i.e. $c{=}t{=}n$) of barrier b is created. We can safely destroy a barrier in its full permission. Waiting on a barrier b requires a unit permission $(1, n)$. This is a contributing factor to certify that there is exactly a predefined number of threads participating in the barrier b. After finishing waiting, the phase number p is increased by 1 and threads proceed to the next phase. The permission rules for split/combine ($\underline{[\textbf{S}-\textbf{SPLIT}]}$ and $\underline{[\textbf{S}-\textbf{COMBINE}]}$) and separation $\underline{[\textbf{S}-\textbf{SEP}]}$ are similar to those of standard bounded permissions.

Our approach allows for local reasoning where each thread (more precisely each procedure) is verified separately. Intuitively, if threads participate in a barrier b, when they join together, their states must agree on the barrier b. Therefore, we enforce the requirement that concurrent threads must maintain a program in barrier-consistent (or *b-consistent*) states:

Bounded permission: $b \xmapsto{c,t} \mathsf{barrier}(p)$

Permission count:	c	Permission invariant:	$0 < c \leq t$
Permission total:	t	Full permission:	$c = t$
Phase number:	p	Partial permission:	$c < t$
		Unit permission:	$c = 1$

Verification rules:

$\{\ n{>}0\ \}$ `barrier b = new barrier(n);` $\{\ b \xmapsto{n,n} \mathsf{barrier}(0)\ \}$

$\{\ b \xmapsto{n,n} \mathsf{barrier}(\_)\ \}$ `destroy(b);` $\{\ \mathsf{emp}\ \}$

$\{\ b \xmapsto{1,n} \mathsf{barrier}(p)\ \}$ `wait(b);` $\{\ b \xmapsto{1,n} \mathsf{barrier}(p+1)\ \}$

Permission rules:

$$\frac{\text{[S-SPLIT]}}{b \xmapsto{c,t} \mathsf{barrier}(p) \wedge c{=}c_1{+}c_2 \implies b \xmapsto{c_1,t} \mathsf{barrier}(p) * b \xmapsto{c_2,t} \mathsf{barrier}(p)}$$

$$\frac{\text{[S-COMBINE]}}{b \xmapsto{c_1,t} \mathsf{barrier}(p) * b \xmapsto{c_2,t} \mathsf{barrier}(p) \implies b \xmapsto{c,t} \mathsf{barrier}(p) \wedge c{=}c_1{+}c_2}$$

$$\frac{\text{[S-SEP]}}{b_1 \xmapsto{c_1,t_1} \mathsf{barrier}(p) * b_2 \xmapsto{c_2,t_2} \mathsf{barrier}(p) \wedge (t_1{\neq}t_2 \ \vee\ c_1{+}c_2{>}t_1) \implies b_1{\neq}b_2}$$

Fig. 6. Verification of Static Barriers

$$\frac{\begin{array}{cc} \{\Phi_1\}\ s_1\ \{\Phi_1^{'}\} & modifies(s_1) \cap FV(\Phi_2, \Phi_2^{'}){=}\varnothing \\ \{\Phi_2\}\ s_2\ \{\Phi_2^{'}\} & modifies(s_2) \cap FV(\Phi_1, \Phi_1^{'}){=}\varnothing \\ \Phi_1 * \Phi_2\ is\ b{-}consistent & \Phi_1^{'} * \Phi_2^{'}\ is\ b{-}consistent \end{array}}{\{\Phi_1 * \Phi_2\}\ s_1 || s_2\ \{\Phi_1^{'} * \Phi_2^{'}\}} \tag{5}$$

Compared with the original rule in (2), our parallel composition rule in (5) additionally requires that concurrent threads begin and end in *b-consistent* states. That is, starting from a consistent state with respect to barriers in the program, threads concurrently operate on the barriers; if they terminate, they do so in a consistent state with respect to the barriers. Informally, a memory state is *b-consistent* if its barrier nodes agree on the phase numbers. After completing their execution, if the threads end up in a joined state $\Phi_1^{'} * \Phi_2^{'}$ which is not b-consistent, the program is rejected as it is incorrectly synchronized. A similar consistency check is also required for the frame rule, which is omitted here since it can be derived from the parallel composition rule (see Section 2.1).

Definition 2 (Combined State) *A combined state Φ_c of a memory state Φ is achieved by repeatedly applying the* $\underline{\text{[S-COMBINE]}}$ *rule until a fixpoint is reached.*

Such a fixpoint always exists as the $\underline{\text{[S-COMBINE]}}$ rule can only reduce the number of heap nodes.

Lemma 2 *A memory state Φ and its combined state Φ_c are equivalent.*

Proof. Φ_c is derived from Φ using $\underline{\text{[S-COMBINE]}}$ rule and Φ can be derived from Φ_c using $\underline{\text{[S-SPLIT]}}$ rule. □

Definition 3 (b-consistency) *A combined state Φ_c is b-consistent iff for every pair of barrier nodes $b_1 \xmapsto{c_1,t_1} \mathsf{barrier}(p_1)$ and $b_2 \xmapsto{c_2,t_2} \mathsf{barrier}(p_2)$ in Φ_c, $b_1{=}b_2 \implies p_1{=}p_2$ holds.*

Corollary 1 *A memory state Φ is b-consistent iff its combined state Φ_c is b-consistent.*

Proof. It directly follows from Lemma 2 as Φ and Φ_c are equivalent. □

Example 1. The memory state $b_1 \xmapsto{1,2} \mathsf{barrier}(p_1) * b_2 \xmapsto{1,2} \mathsf{barrier}(p_1)$ is b-consistent. However, the memory state $b_1 \xmapsto{1,2} \mathsf{barrier}(p_1) * b_2 \xmapsto{1,2} \mathsf{barrier}(p_1{+}1)$ is not since, intuitively, it is possible for $b1$ and $b2$ to be aliased and thus the two aliased barrier nodes have inconsistent phase numbers on the same barrier.

We apply our approach to verification of the programs presented in Fig. 5. The program in Fig. 5a can be proven correctly synchronized. When verifying the program in Fig. 5b, our verification system reports a failure when joining the two threads because the joined state is not b-consistent.

```
                              { emp }
                    barrier b = new barrier(2);
                      { $b \xmapsto{2,2} \mathsf{barrier}(0)$ }
 ( { $b \xmapsto{1,2} \mathsf{barrier}(0)$ }                 || { $b \xmapsto{1,2} \mathsf{barrier}(0)$ }                 )
 ( int i=0;                                 || int j=0;                                )
 ( { $b \xmapsto{1,2} \mathsf{barrier}(0) \wedge i = 0$ }         || { $b \xmapsto{1,2} \mathsf{barrier}(0) \wedge j = 0$ }         )
 ( while (i<10){ wait(b);i++;}              || while (j<20){wait(b); j++; }            ) ;
 ( i=0;                                     ||                                         )
 ( while (i<10){ wait(b);i++;}              ||                                         )
 ( { $b \xmapsto{1,2} \mathsf{barrier}(20) \wedge i = 10$ }       || { $b \xmapsto{1,2} \mathsf{barrier}(20) \wedge j = 20$ }       )
                      { $b \xmapsto{2,2} \mathsf{barrier}(20)$}
```

Fig. 7. More Complex Example

Fig. 7 shows another example which is rather complex due to intricate phasing. Our bounded permissions ensure that there are exactly two threads participating in the barrier `b` while the phase numbers capture exact phasing. Although the two threads operate in different while loops, our notion of phase numbers can certify that the two threads participate in the same numbers of phases. Therefore, the program is correctly synchronized. Our approach is also capable of verifying programs with more intricate sharing and nested fork/join (see Appendix B for such an example program).

5 Verification of Dynamic Barriers

This section presents our approach to verifying correct synchronization of dynamic barriers. In contrast to static barriers whose number of participants is fixed, dynamic barriers allow the number of participants to be changed during a program's execution. For example, .NET framework allows threads to add and remove m participants to/from a barrier b dynamically via **add**(b, m) and **remove**(b, m).[1] We first present a variant of bounded permissions (called *dynamic bounded permissions*) to keep track of the additions and/or removals of

[1] .NET indeed uses AddParticipants() and RemoveParticipants(); we write add() and remove() for brevity.

barrier participants of each thread. We then introduce a set of verification and permission rules to reason about dynamic behaviors of dynamic barriers.

A summary of our approach is presented in Fig. 8. Compared to the bounded permission in Section 3, a dynamic bounded permission of a barrier $b \xmapsto{c,t,a} \mathsf{barrier}(p)$ adds an additional component a, called *permission addition*, to keep track of the additions and/or removals of barrier participants issued by each thread. Permission addition a is a rational number since when splitting a dynamic bounded permission, we require that the split-off permissions have proportional shares of a (details to be presented soon). We also introduce the notion of *zero permission* to capture the fact that a thread has dropped its participation to a barrier (c=0) but still retained its information about the addition and/or removals of participants. Our approach guarantees that zero permission can only be achieved by a thread deliberately removing its participation and cannot be produced by a permission split. A permission quantity (c, t, a) statically captures the local view of a thread on the barrier. With the presence of permission addition a, the full permission is achieved when $c = t + a$. Intuitively, the current number of participants is equal to the original number of participants plus the number of participants added or removed. One could recognize that dynamic bounded permission and bounded permission coincide when a=0.

The verification rules in Fig. 8 capture dynamic behaviors of dynamic barriers. Creating a new barrier results in a full permission of the barrier with a=0. Destroying a barrier requires a full permission (c=t+a). Waiting at a barrier requires a unit permission (c=1). Adding and removing m participants add and respectively subtract m from the permission count and the permission addition. The permission total t remains unchanged; it acts as a pivot for combining permissions when threads join together. A thread can only remove up to the permission count it has ($c \geq m$). If c=m, after removing, a thread is considered dropping its participation to the barrier. Adding participants requires c>0 to ensure that a drop-out thread could not re-participate in a barrier. This is necessary because when dropping out, a thread has lost phasing with other participants; therefore, it is unsafe to allow it to re-participate.

Due to the nature of dynamic barriers, a thread could either fully participate in a barrier (i.e. it does not drop out) or drop its participation in the middle of its execution. Permission rules in Fig. 8 capture those dynamic behaviors. The rule [**D−SPLIT**] never splits into zero permissions; therefore, it ensures that a zero permission only appears due to a thread's drop-out. The rule also ensures that a full permission is never created by splitting a partial permission since it requires that the two split-off permissions have proportional shares of a; that is $a_1 = \frac{c_1}{c} \cdot a$ and $a_2 = \frac{c_2}{c} \cdot a$. We provide the proof for this claim in Appendix E. When multiple threads join, some of them have fully participated in the barrier b while others might drop out midway. Therefore, the combine rules have to take into consideration several situations. First, combining two fully participating threads ($c_1 \neq 0$ and $c_2 \neq 0$) adds up their permission counts and permission additions ([**D−COMBINE−1**]). Because of their full participation, their phase numbers should be equal (both are p). Second, in order to combine one fully-participating

Dynamic bounded permission: $b \xmapsto{c,t,a} \mathsf{barrier}(p)$

Permission count:	c	Permission invariant:	$0 \le c \le t+a$
Permission total:	t	Full permission:	$c = t+a$
Permission addition:	a	Partial permission:	$0 < c < t+a$
Phase number:	p	Unit permission:	$c = 1$
		Zero permission:	$c = 0$

Verification rules:

$\{n{>}0\}$	`b = new barrier(n);`	$\{b \xmapsto{n,n,0} \mathsf{barrier}(0)\}$
$\{b \xmapsto{c,t,a} \mathsf{barrier}(\_) \wedge c{=}t{+}a\}$	`destroy(b);`	$\{\mathsf{emp}\}$
$\{b \xmapsto{1,t,a} \mathsf{barrier}(p)\}$	`wait(b);`	$\{b \xmapsto{1,t,a} \mathsf{barrier}(p+1)\}$
$\{b \xmapsto{c,t,a} \mathsf{barrier}(p) \wedge c{>}0 \wedge m{>}0\}$	`add(b,m);`	$\{b \xmapsto{c+m,t,a+m} \mathsf{barrier}(p)\}$
$\{b \xmapsto{c,t,a} \mathsf{barrier}(p) \wedge c{\ge}m \wedge m{>}0\}$	`remove(b,m);`	$\{b \xmapsto{c-m,t,a-m} \mathsf{barrier}(p)\}$

Permission rules:

[D−SPLIT]

$$b \xmapsto{c,t,a} \mathsf{barrier}(p) \wedge 0{<}c{\le}t{+}a \wedge 0{<}c_1{<}t{+}a_1 \wedge 0{<}c_2{<}t{+}a_2 \wedge c{=}c_1{+}c_2 \wedge a{=}a_1{+}a_2$$
$$\wedge\ a_1{=}\tfrac{c_1}{c}{\cdot}a \wedge a_2{=}\tfrac{c_2}{c}{\cdot}a \implies b \xmapsto{c_1,t,a_1} \mathsf{barrier}(p) * b \xmapsto{c_2,t,a_2} \mathsf{barrier}(p)$$

[D−COMBINE−1]

$$b \xmapsto{c_1,t,a_1} \mathsf{barrier}(p) * b \xmapsto{c_2,t,a_2} \mathsf{barrier}(p) \wedge c_1{\neq}0 \wedge c_2{\neq}0$$
$$\implies b \xmapsto{c,t,a} \mathsf{barrier}(p) \wedge c{=}c_1{+}c_2 \wedge a{=}a_1{+}a_2$$

[D−COMBINE−2]

$$b \xmapsto{c_1,t,a_1} \mathsf{barrier}(p_1) * b \xmapsto{c_2,t,a_2} \mathsf{barrier}(p_2) \wedge c_1{\neq}0 \wedge c_2{=}0 \wedge p_2{\le}p_1$$
$$\implies b \xmapsto{c,t,a} \mathsf{barrier}(p_1) \wedge c{=}c_1{+}c_2 \wedge a{=}a_1{+}a_2$$

[D−COMBINE−3]

$$b \xmapsto{c_1,t,a_1} \mathsf{barrier}(p_1) * b \xmapsto{c_2,t,a_2} \mathsf{barrier}(p_2) \wedge c_1{=}0 \wedge c_2{=}0$$
$$\implies b \xmapsto{0,t,a} \mathsf{barrier}(p) \wedge a{=}a_1{+}a_2 \wedge p{=}max(p_1,p_2)$$

[D−FULL]

$$b \xmapsto{c,t,a} \mathsf{barrier}(p) \wedge c{=}t{+}a \wedge a{\neq}0 \wedge c{>}0 \implies b \xmapsto{c,t+a,0} \mathsf{barrier}(p)$$

[D−SEP]

$$b_1 \xmapsto{c_1,t_1,a_1} \mathsf{barrier}(p_1) * b_2 \xmapsto{c_2,t_2,a_2} \mathsf{barrier}(p_2) \wedge (t_1{\neq}t_2 \ \vee\ c_1{+}c_2{>}t_1{+}a_1{+}a_2)$$
$$\implies b_1{\neq}b_2$$

Fig. 8. Verification of Dynamic Barriers

thread ($c_1{\neq}0$) and a drop-out ($c_2{=}0$), the phase number of the latter is at most that of the former ([D−COMBINE−2]). Intuitively, if a thread has dropped its participation in the middle of an execution, it did not participate in some later phases; therefore, its phase number is at most that of a fully-participating thread. Lastly, combining two drop-outs ($c_1{=}0$ and $c_2{=}0$) retains their total number of additions/removals ($a{=}a_1{+}a_2$) and picks up the maximum between their phase numbers ([D−COMBINE−3]). The rule [D−FULL] reshuffles the full permission into an equivalent form. The rule [D−SEP] introduces the notion of separation in the context of dynamic bounded permissions.

Similar to static barriers, in order to ensure correct synchronization of dynamic barriers and to support local reasoning, our approach requires that concurrent threads maintain a program in dynamic-barrier-consistent (*db-consistent*) states. *Db-consistency* is mostly similar to *b-consistency*; it additionally considers the cases where the phase numbers of barrier nodes of the same barrier are not the same (due to the removal of participants). Due to space limitation, we refer interested readers to Appendix C for more details.

```
1                                 { emp }
2                      barrier b = new barrier(2);
3                       { b ↦^{2,2,0} barrier(0) }
4                              //[D-SPLIT]
5   { b ↦^{1,2,0} barrier(0) }  ||                 { b ↦^{1,2,0} barrier(0) }
6   wait(b);                    ||                          wait(b);
7   { b ↦^{1,2,0} barrier(1) }  ||                 { b ↦^{1,2,0} barrier(1) }
8                               ||                          add(b,1);
9                               ||                 { b ↦^{2,2,1} barrier(1) }
10                              ||                        //[D-SPLIT]
11                              ||  (                             ||
12                              ||  { b ↦^{1,2,1/2} barrier(1) }  ||  { b ↦^{1,2,1/2} barrier(1) }
13  wait(b);                    ||  wait(b);                      ||  remove(b,1);
14  { b ↦^{1,2,0} barrier(2) }  ||  { b ↦^{1,2,1/2} barrier(2) }  ||
15                              ||  remove(b,1);                  ||
16                              ||                                ||
17  wait(b);                    ||  { b ↦^{0,2,-1/2} barrier(2) } ||  { b ↦^{0,2,-1/2} barrier(1) } )
18                              ||                 //[D-COMBINE-3]
19  { b ↦^{1,2,0} barrier(3) }  ||                { b ↦^{0,2,-1} barrier(2) }
19                          //[D-COMBINE-2]
20                      { b ↦^{1,2,-1} barrier(3) }
21                              destroy(b);
22                                { emp }
```

Fig. 9. An Example of Verifying Synchronization of Dynamic Barriers

Fig. 9 presents the proof outline of a program with dynamic barriers. The leftmost thread fully participates in b while the right thread participates in one phase, then adds another participant (line 8), and creates two child threads operating on b. The left child thread drops out after one phase while the right child thread drops out without participation. At the end of the parallel compositions, the permissions are combined together into a full permission. In our approach, for local reasoning, each thread is verified separately and is unaware of operations (such as add/remove) performed by other threads until they join together. Although sound (as proven in Section 6), our approach is incomplete since it could reject programs that are correct at run-time. However, we believe that our static verification is generally a good practice for programmers to follow in order to avoid unexpected run-time behaviors (see Appendix D for details).

6 Soundness

We show that our proposed approach guarantees correct synchronization of dynamic barriers. As dynamic barriers are more general than static barriers, the soundness also implies correct synchronization of static barriers. We first present an encoding of join operations in terms of barrier operations. This encoding simplifies the proof rules and soundness arguments to only focusing on barrier operations. We then proceed to the main soundness arguments of our approach.

Lemma 3 (Soundness of Verifying Barrier Synchronization) *Given a program with a barrier b and a set of procedures P^i together with their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, then the program is correctly synchronized with respect to the barrier b.*

Proof. Detailed definitions and proofs can be found in Appendix E. □

7 Implementation and Experimental Results

We implemented our approach into a prototype tool, named VeriBSync[2]. We applied VeriBSync to verifying static[3] barrier synchronization of all twelve simplified[4] programs of SPLASH-2 suite [25]. SPLASH-2 suite is one of the most widely used benchmarks for evaluating shared-memory systems. The suite consists of twelve realistic programs covering numerous application domains such as computer graphics (`volrend`), signal processing (`fft`), water molecule simulation (`water-spatial`), and general engineering (`radix`) among others. Besides the theoretical contributions, the empirical question we investigate is how well our approach handles realistic barrier synchronization. The results were promising as our approach was able to verify all but one program in SPLASH-2 suite with modest annotation. All experiments were done on a 3.20GHz Intel Core i7-960 processor with 16GB memory running Ubuntu Linux 10.04. The suite of benchmark programs and other examples are provided in our project website.

The experimental results are presented in Table 1. The column *#Bar* shows the number of barriers used in the corresponding program. The column *LOC* shows the total number of non-blank, non-comment, non-annotation lines of source code, counted by `sloccount` (v2.26). The column *LOAnn* shows the total number lines of annotation. Annotation *overhead* is computed as $\frac{LOAnn}{LOC}$ (the lower, the better). Verification times are in seconds. VeriBSync was able to verify barrier synchronization of all but one program in SPLASH-2 suite with the verification time in several seconds. We discuss the reason why VeriBSync was

[2] The tool is available for both online use and download at `http://loris-7.ddns.comp.nus.edu.sg/~project/veribsync/`.

[3] As dynamic barriers have just been available recently since .NET 4.0 (April 2010) and Java 7 (July 2011), we are not aware of existing concurrency benchmarks that use dynamic barriers. Nonetheless, we applied our prototype on a set of textbook programs which represent typical usage of dynamic barriers. The programs are available in our project website.

[4] As verifying full functional correctness of these programs is beyond the scope of this paper, our experiments were conducted on a set of simplified programs where parts of programs that are not related to barriers were omitted. All related parts such as branching conditions and loops were retained to ensure that barrier synchronizations in the simplified programs are similar to those of the original programs.

Table 1. Annotation Overhead and Verification Time of SPLASH-2 Suite

Program	Description	#Bar	LOC	LOAnn	Overhead	Time
`ocean`	large-scale ocean simulation	1	60	5	8%	1.01
`radix`	integer radix sort	2	68	7	10%	3.11
`lu`	blocked LU decomposition	1	79	12	15%	14.33
`barnes`	Barnes-Hut for N-body problem	1	84	12	14%	2.35
`raytrace`	optimized ray tracing	1	94	7	7%	0.44
`fft`	complex 1D FFT	1	101	8	8%	0.69
`water-nsquared`	water simulation w/o spatial data structure	3	113	16	14%	13.23
`water-spatial`	water simulation w/ spatial data structure	3	117	18	15%	13.53
`cholesky`	blocked sparse cholesky factorization	1	131	10	8%	0.50
`fmm`	adaptive fast multipole for N-body problem	1	175	20	11%	0.79
`volrend`	optimized ray casting	2	232	36	16%	7.50
`radiosity`	hierarchical diffuse radiosity method	1	83	-	-	-
Average	-	-	-	-	11%	5.23

not able to verify `radiosity` program in Section 8.1. The verification time and annotation overhead depend on characteristics of the programs. Programs that have complicated non-linear constraints and/or use barriers in many execution branches (such as `lu`, `barnes`, `water-*`, and `volrend`) require higher verification time and annotation overhead. On average, VeriBSync requires annotation overhead of 11%, which is modest compared with that of 100% reported in the literature [11].[5] Much of the annotation and verification time are dedicated for functional correctness properties of the programs such as branching conditions and loops. As annotation efforts for these properties are also necessary for verifying functional correctness of concurrent programs, we believe that existing logics for verifying functional correctness can easily integrate our approach into their logics and benefit from our guarantee of correct barrier synchronization.

8 Discussion

This section discusses limitations and future extensions of our existing approach.

8.1 Functional Correctness vs. Barrier Synchronization

In our existing approach, threads are correctly synchronized if they end up with the same (determinable) phase numbers. However, there are programs (such as `radiosity`) where the phase numbers are tightly coupled with functional correctness. A fragment of `radiosity` program is shown in Fig. 10. The barrier `barrier` is used within the while loop which terminates only when the solution converges (by calling the procedure `init_ray_tasks` to check for convergence). The `init_ray_tasks` procedure only allows one thread (the first thread entering) to check for convergence and to update a global variable while other threads only read that variable. Such barrier phasing, therefore, is deeply correlated with functional correctness of the program (i.e. the convergence) which could not be captured by our existing approach. However, our approach could be extended to verify this type of programs by considering resource re-distribution that could be used to verify functional correctness, and

```
/* ... perform ray-gathering till
the solution converges */
while( init_ray_tasks(...) ) {
  wait(barrier);
  process_tasks(...);
}
```

Fig. 10. A Fragment of `radiosity`

[5] To be precise, the annotation overhead in [11] also includes the specification for functional correctness. Although verifying functional correctness is not our main goal, we also need to specify them for verifying barrier synchronization.

the use of existential phase numbers. Details of such an extension will be more carefully investigated in the near future.

8.2 Deadlock-free Multiple Barriers

Correct synchronization is a property weaker than deadlock freedom: it ensures deadlock freedom in case of a single barrier. When using multiple barriers, their synchronization patterns could potentially lead to deadlocks. We plan to extend our existing approach with barrier expressions to capture patterns of participating in multiple barriers. Together with the phase numbers, by proving that the barrier expressions of different participants are compatible, we could guarantee deadlock freedom. Patterns of participating in multiple barriers have been used in verification of SPMD programs with static barriers [2, 15, 27]. However, adapting them to verification of fork/join programs with dynamic barriers is non-trivial. This is not only because we need to address the unstructured nature of fork/join programs (in SPMD programs, threads execute the same code while in fork/join programs, they execute different pieces of code), but also because we need to handle dynamic allocation/deallocation and addition/removals of participants in a modular way. We leave this topic for future investigation.

9 Related Work

This section discusses related works regarding access permission systems and static verification of barrier synchronization. We also discuss related works regarding other advanced forms of barriers such as X10's clocks [22] and phasers [23] which have recently been introduced in the context of async/finish programs.

9.1 Access Permissions

Boyland first introduced fractional permissions for reasoning about non-interference of concurrent programs [5]. Bornat et. al. added counting permissions [4]. Recently, various permission systems such as binary tree share model [6], Plaid's permission system [3], and borrowing permissions [19] have been proposed. In a nutshell, they are akin to fractional and counting permissions.

Importantly, not every program is suitable for fractional permissions and counting permissions. Programs that allow sharing resources among only a bounded number of threads need another alternative treatment. Fractional and counting permissions could not reason about those programs because, when using these permission systems, there is nothing to prevent a resource from being split off an unbounded number of times and shared among an unbounded number of threads. Given any fractional permission f where $0{<}f{\leq}1$, it is always possible to split f into two fractions f_1 and f_2 where $f_1{+}f_2{=}f$ and $f_1, f_2{>}0$. Similarly, in counting permissions, given a central *permission authority* holding a source permission n, it is always possible to split off into a new source permission $n{+}1$ (held by the central authority) and a read permission -1 for sharing. On the other hand, in our bounded permission system, any non-unit permission (c, t) where $1{<}c{\leq}t$ (either partial or full permissions) can be split off without the presence of a central authority, and a bounded permission can only be split off a bounded number of times (up to unit permissions). Therefore, bounded permissions enable reasoning about bounded resources such as barriers.

9.2 Verification of Barrier Synchronization

Most existing works on verifying barrier synchronization focus on SPMD programs [2, 12, 14–16, 27, 26]. In SPMD programs, the fact that threads execute the same code makes verification more tractable. SMPD programs also assume that barriers are global and all threads need to participate in barrier operations. Hence, existing techniques for SPMD programs cannot be directly applied to fork/join programs. This paper fills in the gap and addresses barriers in the context of fork/join concurrency where concurrent threads could execute different pieces of code while participating in barrier operations. Furthermore, we do not restrict that all threads should participate, i.e. a group of threads can participate on a certain barrier. We also support verification of dynamic barriers whose number of participants can vary during a program's execution. We are not aware of any related works capable of verifying dynamic barriers in fork/join programs.

To the best of our knowledge, the most closely related work is by Hobor and Gherghina [11]: they propose a specification logic for verifying partial correctness of programs with static barriers. Based on the global *phase transition specification* of a barrier, they can also verify that participants proceed in correct phases. However, there are several critical differences. First, they do not handle dynamic barriers. Second, they require a global specification of each barrier, whereby programmers have to specify pre-state and post-state for each thread for every phase transition over the barrier. However, there are programs (such as that in Fig. 7) where our approach using phase numbers can verify, but it is not possible to capture a global specification for its barrier [8]. Though the global specification of each barrier is an extra annotation burden, they can facilitate resource re-distribution at synchronization points to ensure functional partial correctness. Our current approach using phase numbers is considerably simpler, but has not yet been designed to support resource re-distribution. This may be important for more complex usage of barrier synchronization.

9.3 Advanced Forms of Barriers

There are various implementations of barriers [10], and several implementations have been verified in [17]. Our specification in Fig. 6 and 8 can serve as a common interface for verifying different implementations. Beside traditional barriers, other advanced forms of barriers X10's clocks [22] and phasers [23] are also used in the context of async/finish programs. Note that Java 7's Phaser [9] only includes a subset of capability of the phasers proposed in [23], i.e. Java 7's Phaser is similar to dynamic barriers used in .NET [7] (which are the main topic of Section 5). Compared with traditional barriers, clocks and phasers are more dynamic in nature and are only applied to the more tractable context of async/finish programs.

Barrier synchronization in async/finish programs is generally more tractable than that in fork/join programs for two main reasons. First, thread creation and join in async/finish programs are lexically-scoped while those in fork/join are non-lexically-scoped, i.e. fork and join operations can be invoked in different program scopes. Second, there are restrictions on the usage of clocks and phasers in async/finish programs [22, 23]. For example, in X10 programs, a newly-spawn thread has to explicitly register and directly operate on a clock, and it can only register to the clock that its parent has already registered to. These restrictions reject many useful programs such as the program presented in Figure 11 of Appendix B. On the other hand, in fork/join programs written in mainstream languages such as C/C++ (with Pthreads), Java, and .NET, there aren't such restrictions. A new thread does not need to register but can still freely own or pass a barrier to other threads. Because of these reasons, one could not directly apply analyses and verification techniques of clocks in async/finish programs (e.g. those in [13, 18]) to traditional barriers in fork/join programs. In contrast, we conjecture that one could adapt our proposed approach to statically verifying correct synchronization of clocks and phasers.

10 Conclusion

We described a specification and verification approach for ensuring correct synchronization of software barriers. Barriers, provided by many mainstream languages such as C/C++ (with Pthreads), Java, and .NET, are hard to handle in fork/join programs because programmers must not only pay special attention to the (possibly dynamic) number of participating threads, but also ensure that threads proceed in correctly synchronized phases. To our knowledge, this is the first work that statically ensures the correct synchronization of both *static* and *dynamic* barriers in fork/join programs. The keys of our approach are the *bounded permissions* and *phase numbers* to keep track of the number of participating threads and barrier phases respectively. Not restricted to only barriers, bounded permissions can be generally used to reason about any resources that are shared among a bounded number of concurrent threads. Our approach has been proven sound, and a prototype of it has been implemented for verifying barrier synchronization of realistic programs in SPLASH-2 benchmark suite.

Acknowledgement. We are grateful to our colleagues and the anonymous reviewers for their comments. Special thanks to C. Gherghina for many insightful discussions, and to H. D. Nguyen for suggesting the use of rational numbers for soundness. This work is supported by MOE Project 2009-T2-1-063.

References

1. The Open Group Base Specifications Issue 7 IEEE Std 1003.1-2008. `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`.
2. A. Aiken and D. Gay. Barrier Inference. In *POPL*, pages 342–354, 1998.
3. K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320, 2007.
4. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission Accounting in Separation Logic. In *POPL*, pages 259–270, New York, NY, USA, 2005. ACM.
5. J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, 2003.
6. R. Dockins, A. Hobor, and A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting. In *APLAS*, pages 161–177, 2009.
7. A. Freeman. *Pro .NET 4 Parallel Programming in C#.* Apress, 2010.
8. C. Gherghina. Personal communication, April 2013.

9. J. F. González. *Java 7 Concurrency Cookbook*. Packt Pub Limited, 2012.
10. J. M. D. Hill and D. B. Skillicorn. Practical Barrier Synchronisation. In *PDP*, pages 438–444, 1998.
11. A. Hobor and C. Gherghina. Barriers in concurrent separation logic: Now with tool support! *Logical Methods in Computer Science*, 8(2), 2012.
12. T. E. Jeremiassen and S. J. Eggers. Static Analysis of Barrier Synchronization in Explicitly Parallel Programs. In *PACT*, pages 171–180, 1994.
13. S. Joshi, R. K. Shyamasundar, and S. K. Aggarwal. A New Method of MHP Analysis for Languages with Dynamic Barriers. In *IPDPS Workshops*, pages 519–528, 2012.
14. A. Kamil and K. A. Yelick. Concurrency Analysis for Parallel Programs with Textually Aligned Barriers. In *LCPC*, pages 185–199, 2005.
15. A. Kamil and K. A. Yelick. Enforcing Textual Alignment of Collectives Using Dynamic Checks. In *LCPC*, pages 368–382, 2009.
16. Y. Lin. Static Nonconcurrency Analysis of OpenMP Programs. In *IWOMP*, 2005.
17. A. Malkis and A. Banerjee. Verification of software barriers. In *PPoPP*, 2012.
18. F. Martins, V. T. Vasconcelos, and T. Cogumbreiro. Types for X10 Clocks. In *PLACES*, pages 111–129, 2010.
19. K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A Type System for Borrowing Permissions. In *POPL*, pages 557–570, 2012.
20. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, 2004.
21. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, Copenhagen, Denmark, July 2002.
22. V. A. Saraswat and R. Jagadeesan. Concurrent Clustered Programming. In *CONCUR*, pages 353–367, 2005.
23. J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer III. Phasers: a Unified Deadlock-free Construct for Collective and Point-to-point Synchronization. In *ICS*, pages 277–288, 2008.
24. J. Smans, B. Jacobs, and F. Piessens. Implicit Dynamic Frames. *TOPLAS*, 2012.
25. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ICSA*, 1995.
26. Y. Zhang and E. Duesterwald. Barrier Matching for Programs with Textually Unaligned Barriers. In *PPoPP*, pages 194–204, 2007.
27. Y. Zhang, E. Duesterwald, and G. R. Gao. Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers. In *LCPC*, 2007.

A Programming Language

In this section, we describe the core fork/join programming language with barriers presented in Section 2.2. A program consists of a list of procedure declarations *proc*. Each procedure *proc* is annotated with pairs of pre/post-conditions (Φ_{pr}/Φ_{po}) written in existing specification logics such as separation logic [21] and implicit dynamic frames [24] enhanced with specification for barriers. The core programming language supports primitive types (such as `int`, `bool`, and `void`), and barrier type `barrier`. Key statements for concurrency consist of fork/join for thread management and barrier operations. A **fork** receives a procedure name `pn` and a list of parameters v^*, spawns a new thread executing the procedure, and returns a unique thread identifier as an integer. A **join** requires a thread identifier to join the thread back. For brevity of presentation, we write the parallel composition $(s_1||s_2)$; as an abbreviation for creating concurrent threads (we sometimes omit (); due to space limit). The parallel composition is just syntactic sugar which can easily be encoded via fork and join. `barrier` $b =$ **new** `barrier`(n) creates a new barrier b with the number of participants n. **destroy**(b) destroys the barrier b. A thread issues an barrier wait by calling **wait**(b). For dynamic barriers, **add**(b, m) and **remove**(b, m) adds and respectively removes m participants from the existing total number of participants. The semantics of other program statements (such as procedure calls `pn`(v^*), conditionals, loops, assignments) are standard as can be found in well-known languages such as C/C++, Java, and .NET.

B Example Program with Intricate Sharing and Nested Fork/Join

Fig. 11 shows an example of programs with more intricate sharing and nested fork/join. Inside `main`, the main thread creates two child threads executing the procedure `group` on two different barriers `b1` and b2. These two threads do not directly operate on their respective barrier but they create two grand-child threads to participate instead. Consequently, permissions of barrier `b1` and `b2` are transferred from the main thread to child threads and finally to the grand-child threads to create two different groups of grand-child threads participating on two different barriers. Based on the phase numbers, we can verify that threads participate in the same numbers of phases. Note that after joining back the child threads, the main thread gets back the full permissions for `b1` and `b2`. Programmers need not indicate the fact that b1 and b2 are different barriers. Verifiers can use our [**S−SEP**] rule to infer that information automatically.

C Consistency Requirements for Dynamic Barriers

Similar to static barriers, in order to ensure correct synchronization of dynamic barriers and to support local reasoning, our approach also requires that concurrent threads maintain a program in dynamic-barrier-consistent (*db-consistent*) states. However, the check for *db-consistency* is slightly more complex than that

```
void main()
  requires emp
  ensures emp;
{
 { emp }
 barrier b1= new barrier(2);
 barrier b2= new barrier(2);
 { $b1 \xrightarrow{2,2}$ barrier(0) * $b2 \xrightarrow{2,2}$ barrier(0) }
 int idg1=fork(group,b1);
 { $b2 \xrightarrow{2,2}$ barrier(0) }
 int idg2=fork(group,b2);
 { emp }
 join(idg1);
 join(idg2);
 { $b1 \xrightarrow{2,2}$ barrier(1) * $b2 \xrightarrow{2,2}$ barrier(1) }
 destroy(b1);destroy(b2);
 { emp }
}
```

```
void participant(barrier b)
  requires $b \xrightarrow{1,n}$ barrier(0)
  ensures $b \xrightarrow{1,n}$ barrier(1);
{ wait(b); }

void group(barrier b)
  requires $b \xrightarrow{2,2}$ barrier(0)
  ensures $b \xrightarrow{2,2}$ barrier(1);
{
 { $b \xrightarrow{2,2}$ barrier(0) }
 int id1=fork(participate,b);
 { $b \xrightarrow{1,2}$ barrier(0) }
 int id2=fork(participate,b);
 { emp }
 join(id1);join(id2);
 { $b \xrightarrow{2,2}$ barrier(1) }
}
```

Fig. 11. Nested Fork/Join

of *b-consistency* because in case of dynamic barriers the phase numbers of different barrier nodes of the same barrier need not be the same (due to the addition and removal of participants). Note that since dynamic barriers subsume static barriers (i.e. when $a = 0$), the definition of *db-consistency* also subsumes that of *b-consistency*

Definition 4 (Combined State) *A combined state Φ_c of a memory state Φ is achieved by repeatedly applying the* [**D−COMBINE−1**]*,* [**D−COMBINE−2**]*, and* [**D−COMBINE−3**] *rules until a fixpoint is reached.*

Lemma 4 *A memory state Φ and its combined state Φ_c are equivalent.*

Proof. Φ_c is derived from Φ using [**D−COMBINE−1**], [**D−COMBINE−2**], and [**D−COMBINE−3**] rules and Φ can be derived from Φ_c using a modified [**D−SPLIT**] rule which allows splitting off zero permissions. □

Definition 5 (db-consistency) *A combined state Φ_c is db-consistent iff for every pair of dynamic barrier nodes $b_1 \xrightarrow{c_1,t_1,a_1}$ barrier(p_1) and $b_2 \xrightarrow{c_2,t_2,a_2}$ barrier(p_2) in Φ_c, the following assertion holds:*

$$b_1{=}b_2 \implies (\ (c_1{\neq}0 \land c_2{\neq}0 \land p_1{=}p_2) \lor (c1{=}0 \land p_1{\leq}p_2) \lor (c2{=}0 \land p_2{\leq}p_1)\ )$$

Corollary 2 *A memory state Φ is db-consistent iff its combined state Φ_c is db-consistent.*

Proof. It directly follows from Lemma 4 as Φ and Φ_c are equivalent. □

D Static Verification versus Run-time Behaviors

Our approach can statically verify that a program is correctly synchronized in the presence of static and dynamic barriers. For local reasoning, each thread is verified separately and has its own view on a barrier b (reflected in the bounded permission of b that it owns). Thus, a thread is unaware of operations (such as add/remove) performed by other threads until they join together. Although sound, our approach is incomplete since it could reject programs that are correct at run-time. For example, our static verification (with local reasoning) does not allow the program in Fig. 12a where the left thread is intended to remove the participation of the right thread. However, we believe that a more desirable way to implement this program is to let the right thread deliberately drop its participation (as depicted in Fig. 12b). This more intuitive coding style is readily captured by our approach.

In many cases, our static verification is helpful for preventing harmful behaviors at run-time such as deadlocks due to inter-thread addition/removal of participants. One example is the program presented in Fig. 13a where the left thread adds one participant to the barrier `b` while the right thread adds one more thread participating in `b`. The programmers' intention is that, after adding one more participant, there will be three threads concurrently operating on the barrier. Unfortunately, the program is potentially deadlocked due to the following interleaving: $1 \mapsto 4 \mapsto 5 \mapsto 6 \mapsto 2 \mapsto 3$. In this interleaving, the left thread waits forever at statement 3 because it has to wait for two other participants to issue a barrier wait, though they have already completed execution. Another example is the program in Fig. 13b where the left thread removes one participant while the right thread concurrently adds one participant. Although the total number of participants remains unchanged, the program is potentially deadlocked due to the interleaving $1 \mapsto 4 \mapsto 2 \mapsto 3 \mapsto 5 \mapsto 6$. Fortunately, such error-prone programs with inter-thread addition/removal of participants are rejected by our approach. We believe that our static verification is generally a good practice for programmers to follow in order to avoid unexpected run-time behaviors.

$\{\mathsf{emp}\}$
`barrier b = new barrier(2);`
$\{b \xmapsto{2,2,0} \mathsf{barrier}(0)\}$

$\{b \xmapsto{1,2,0} \mathsf{barrier}(0)\}$	$\{b \xmapsto{1,2,0} \mathsf{barrier}(0)\}$
`wait(b);`	`wait(b);`
$\{b \xmapsto{1,2,0} \mathsf{barrier}(1)\}$	$\{b \xmapsto{1,2,0} \mathsf{barrier}(1)\}$
$\boxed{\texttt{remove(b,1);}}$	`//no longer`
$\{b \xmapsto{0,2,-1} \mathsf{barrier}(1)\}$	`//participate`
`wait(b); //FAIL`	

. . .

(a) Rejected

$\{\mathsf{emp}\}$
`barrier b = new barrier(2);`
$\{b \xmapsto{2,2,0} \mathsf{barrier}(0)\}$

$\{b \xmapsto{1,2,0} \mathsf{barrier}(0)\}$	$\{b \xmapsto{1,2,0} \mathsf{barrier}(0)\}$
`wait(b);`	`wait(b);`
$\{b \xmapsto{1,2,0} \mathsf{barrier}(1)\}$	$\{b \xmapsto{1,2,0} \mathsf{barrier}(1)\}$
`wait(b);`	$\boxed{\texttt{remove(b,1);}}$
$\{b \xmapsto{1,2,0} \mathsf{barrier}(2)\}$	$\{b \xmapsto{0,2,-1} \mathsf{barrier}(1)\}$

$\{b \xmapsto{1,2,-1} \mathsf{barrier}(2)\}$

. . .

(b) Verified

Fig. 12. Dynamic Behaviors of Dynamic Barriers

```
          barrier b = new barrier(2);
 1: wait(b);      || 4:      wait(b);
 2: add(b,1);     ||
 3: wait(b);      ||  5: wait(b); || 6: wait(b);
```

$1 \mapsto 4 \mapsto \mathbf{5} \mapsto \mathbf{6} \mapsto \mathbf{2} \mapsto \mathbf{3}$ (Deadlocked)

(a)

```
          barrier b = new barrier(2);
 1: wait(b);        ||   4: wait(b);
 2: remove(b,1);    ||   5: add(b,1);
 3: wait(b);        ||   6: wait(b);
```

$1 \mapsto 4 \mapsto \mathbf{2} \mapsto \mathbf{3} \mapsto \mathbf{5} \mapsto \mathbf{6}$ (Deadlocked)

(b)

Fig. 13. Potential Deadlocks due to Inter-thread Addition/Removal of Participants

E Soundness

We first present an encoding of join operations in terms of barrier operations. This encoding simplifies the proof rules and soundness arguments to only focusing on barrier operations. We then proceed to the soundness arguments of our verification approach. Note that our approach currently does not consider non-termination due to infinite loops/recursion or deadlocks.

E.1 Encoding of Join Operations

Join operations can be encoded via barriers. Intuitively, each forked procedure receives an extra parameter `b` of type `barrier` and a unit permission to wait on that barrier. Before forking a child thread, a new barrier with two participants is created and passed to the child thread. The child thread will wait on that barrier before it terminates. A thread can join another thread by waiting on the corresponding barrier of the latter.

We present details of the encoding. Given a forked procedure *pn* which is defined as $\mathtt{pn}(\mathtt{w}_1, \ldots, \mathtt{w}_\mathtt{n})$ **requires** Φ_{pr} **ensures** Φ_{po}; { `s` }, we (1) add one more parameter b of type `barrier` to its list of its parameters, (2) add a barrier wait at the end of the procedure, and (3) modify its specification as follows:

$\mathtt{pn}(\mathtt{w}_1, \ldots, \mathtt{w}_\mathtt{n}, \mathtt{b})$

requires $\Phi_{pr} * b \xmapsto{1,2} \mathsf{barrier}(0)$

ensures $\Phi_{po} * b \xmapsto{1,2} \mathsf{barrier}(1)$;

{ `s;  wait(b);` }

Then, we encode `int id=fork(pn,w₁, ..., wₙ);` as `barrier b = new barrier(2); int id=fork(pn,w₁, ..., wₙ);` and encode `join(id)` as `wait(b)`. It is easy to see that the encoding results in correct synchronization of the newly added barrier `b`: two threads (the forker and the forkee) have unit permissions to access `b` and they both wait on `b` just once.

E.2 Soundness of Dynamic Bounded Permissions

We prove that, besides boundedness, our dynamic bounded permission system exercises properties of a standard access permission system: it allows concurrent reads and exclusive write. That is, we prove that, when using our verification and permission rules in Fig. 8, splitting and combining from any partial permissions never result in a full permission unless all partial permissions of b are combined. In this section, for brevity, we often refer to a permission $b \xmapsto{c,t,a} \mathsf{barrier}(p)$ by its quantity (c, t, a).

Let S_b and t_b denote the set of all partial permissions and respectively the permission total of a barrier b.

Corollary 3 (Full Permission) *Combining all partial permissions of a barrier b results in a full permission of b.*

Proof. First, the permission total t_b of a barrier b can only be safely changed by the rule [**D−FULL**]. Otherwise, t_b remains unchanged under the rest of permission rules and verification rules in Fig. 8. Hence, we would like to prove that $\sum_{(c_i,\_,\_)\in S_b} c_i = t_b + \sum_{(\_,\_,a_i)\in S_b} a_i$ holds. We prove it by induction on the verification and permission rules. The equality trivially holds when the barrier b is created. Destroy and wait operations does not affect the quantity of permissions. Add and remove operations add and respectively subtract the same amount to/from c and a of a barrier node, hence the equality holds under the operations. All permission rules also maintain the equality. □

Corollary 4 (Permission Invariant) $\forall (c, t_b, a) \in S_b\ ,\ c{>}a.$

Proof. The invariant $c{>}a$ trivially holds when a barrier b is created. Destroy and wait operations does not affect the quantity of permissions. Add and remove operations add and respectively subtract the same amount to/from c and a of a barrier node, hence the invariant holds under the operations.

We prove that split/combine rules also maintain the invariant.

For the rule [**D−SPLIT**], we have:

- $c{>}a$ or $\frac{a}{c}{<}1$
- $a_1{=}\frac{c_1}{c}{\cdot}a$ and $a_2{=}\frac{c_2}{c}{\cdot}a$

Hence, we conclude that $c_1{>}a_1$ and $c_2{>}a_2$.

For the combine rules [**D−COMBINE−1**], [**D−COMBINE−2**], and [**D−COMBINE−3**], we have:

- $c_1{>}a_1$ and $c_2{>}a_2$
- $c{=}c_1{+}c_2$ and $a{=}a_1{+}a_2$

Hence, we conclude that $c{>}a$. □

Lemma 5 (Soundness of Dynamic Bounded Permission) *Given a barrier b, our approach ensures that splitting and combining from any partial permissions of b never result in a full permission unless all partial permissions of b are combined.*

Proof. First, it follows from Corollary 3 that combining all partial permissions in S_b resulting in a full permission of b. We then show that it is impossible to combine a strict subset of S_b into a full permission of b.

Assume there exists a strict subset S of all partial permissions of b such at combining partial permissions in S results in a full permission of b. We have $S \subset S_b$. We define $\bar{S}$ the set of partial permissions of b not in S, that is $S_b = S \cup \bar{S}$.

Combining all permissions in S_b results in a full permission:

$$\sum_{(c_i,\_,\_)\in S_b} c_i \quad = \quad t_b \quad + \sum_{(\_,\_,a_i)\in S} a_i \tag{6}$$

As $S_b = S \cup \bar{S}$ and (6), we have:

$$\sum_{(c_k,\_,\_)\in S} c_k \quad + \quad \sum_{(c_j,\_,\_)\in \bar{S}} c_j \quad = \quad t_b \quad + \quad \sum_{(\_,\_,a_k)\in S} a_k \quad + \quad \sum_{(\_,\_,a_j)\in \bar{S}} a_j \tag{7}$$

Combining permissions in S also results in a full permission:

$$\sum_{(c_k,\_,\_)\in S} c_k \quad = \quad t_b \quad + \sum_{(\_,\_,a_k)\in S} a_k \tag{8}$$

From (7) and (8), we have the equality:

$$\sum_{(c_j,\_,\_)\in \bar{S}} c_j \quad = \quad \sum_{(\_,\_,a_j)\in \bar{S}} a_j \tag{9}$$

This contradicts with Corollary 4 as $c>a$ forall (c, t_b, a); hence $\sum_{(c_j,\_,\_)\in \bar{S}} c_j \quad > \quad \sum_{(\_,\_,a_j)\in \bar{S}} a_j$. □

E.3 Soundness of Verifying Barrier Synchronization

We first define what it means for a program to be correctly synchronized with respect to a dynamic barrier.

Definition 6 (Compatible Phasing) *Given a dynamic barrier b with the last phase p (also called final phase), a thread is said to operate on b in a compatible number of phases* p_1 *iff:*

- *If it fully participates in b (i.e. it does not drop out), then* $p1{=}p$.
- *If it drops out, then* $p1{\leq}p$.

Definition 7 (Correct Dynamic Synchronization) *A program is correctly synchronized with respect to a dynamic barrier b iff:*

- *There is exactly a predefined number of threads participating in the barrier b's wait operations.*
- *Participating threads operate on b in compatible numbers of phases.*

Note that in case of static barriers, threads are not allowed to drop out. Therefore, compatible phasing implies that all participants fully participate and operate in the same numbers of phases.

A thread can be in one of four states: *running*, *waiting*, *done*, and *aborted*. Our verification approach ensures that no thread reaches an *aborted* state. A program state is *non-aborting* if neither of threads are in an *aborted* state. A program state is *final* if all threads are in a *done* state.

Definition 8 (Thread State) *A thread state σ is one of the following states:*

- $\mathbf{run}(s, \Gamma)$ *stating that the thread is running with remaining statement s and environment Γ. For brevity, Γ is assumed to be a partial function from object names to object references and from stack variables to values. Environment Γ resembles stack and heap in programs.*
- $\mathbf{wait}(o, s, \Gamma)$ *stating that the thread is waiting at barrier object o with remaining statement s and environment Γ.*
- **done** *stating that a thread has completed its execution.*
- **aborted** *stating a thread has performed an illegal operation.*

Threads in a program wait at barrier points and proceed in phases. We distinguish between local phase and global phase of a barrier. When a participant reaches a barrier point, it increments its local phase. When all participants have reached that point, the global phase will be incremented. If a thread still participates in a barrier, its local phase is at most one ahead of the global phase. Intuitively, after reaching a barrier point and incrementing its local phase, a participant can only proceed if its local phase is equal to the global phase. This semantics has the advantage that a participant only needs to know its local phase and the global phase without worrying about the phases of other participants.

Definition 9 (Program State) *A program state Ψ consists of:*

- *G representing a partial function from barrier objects to tuples (i, t, p) where i is the number of participants that have been suspended (i.e. waiting to proceed to the next phase), t is the total number of participants, and p is the current global phase of barrier object o. We write $G_i(o)$, $G_t(o)$, and $G_p(o)$ denote i, t, and p respectively. A barrier object o is already allocated if $o \in dom(G)$.*
- *T representing a set of threads. Each thread is a tuple (ι, σ, L) consisting of thread identifier ι, thread state σ, and a local barrier map L. L maps barriers to their corresponding local phases.*

Definition 10 (Execution) *Execution of a program starts in the initial program state:* $(\ \emptyset,\ (\_, \mathbf{run}(s, \emptyset), \emptyset)\ )$.

Small-step operational semantics is presented in Fig. 14. In the figure, $def(pn)$ denotes the definition of the procedure pn in the program, $eval(e, \Gamma)$ denotes the evaluation of the expression e in the environment Γ. A premise marked with light gray indicates conditions that need to hold, otherwise the thread has performed an illegal operation and it transitions to an aborted state. For example, a thread adds or removes to/from a barrier a negative number of participants. Our verification rules ensure that the premises in light gray hold, i.e. threads cannot transition to aborted states.

Most of the rules in Fig. 14 are straightforward. When forking a new child thread, the main thread passes the global phase to the child thread. The treatment of loops is similar to that of if-then-else and is omitted. When issue a barrier wait, a thread transitions to a waiting state. The final thread issuing a barrier wait increments the global phase p by 1 and resets the counter i to 0. Threads transition back to a running state when all participants have issued a barrier wait, i.e. the global phase is equal to threads' local phases.

Lemma 6 (Correct Participation) *Given a program with a barrier b and a set of procedures P^i together with their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, then there is exactly a predefined number of threads participating in b's wait operations.*

Proof. Our verification rules rely on bounded permissions to handle concurrent accesses to barrier b. Given n is the predefined number of participants, it follows from Lemma 1 that there are at most n threads concurrently operating on barrier b. In order to perform a wait on barrier b, threads must have unit permissions of barrier b. Additionally, adding and removing participants correspond to the addition and subtraction of the total number of participants t in operational semantics, i.e. $t_b + \sum_{(c_i,t_b,a_i)\in S_b} a_i = t$ where t_b is the original number of participants declared at b's creation point. Hence, there are exactly n threads participating in barrier b. □

Lemma 7 (Correct Phasing) *Given a program with a barrier b and a set of procedures P^i together with their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, then threads participating on barrier b operate in compatible numbers of phases.*

Proof. The phase number used in our barrier specification corresponds to the local phase in the operational semantics. The final phase of b corresponds to the global phase of b after all participants complete their execution. First, if a thread fully participates on barrier b (it does not drop out), then it ends up in a local phase which is equal to the global phase. Second, if a participant drops out, it ends up in a local phase which is at most equal to the global phase. Third, if a thread does not fully participate on barrier b, does not drop out, and ends up in a phase which is not the final phase, it will be rejected by the db-consistency check (See Appendix C). Hence, all participants end up in compatible numbers of phases. □

Lemma 8 (Soundness of Verifying Barrier Synchronization) *Given a program with a barrier b and a set of procedures P^i together with their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, then the program is correctly synchronized with respect to the barrier b.*

Proof. It directly follows from Lemma 6, Lemma 7, and Definition 7. □

$$\frac{\begin{array}{c} o \notin \mathit{dom}(G) \qquad \mathit{typeof}(o) = \texttt{barrier} \qquad \mathit{\Gamma}(n) = \mathit{num} \qquad \mathit{num}{>}0 \\ \mathit{\Gamma}' = \mathit{\Gamma}[b \mapsto o] \qquad G' = G[o \mapsto (0, \mathit{num}, 0)] \qquad L' = L[o \mapsto 0] \end{array}}{\begin{array}{c} (G, \{(\iota, \texttt{b = new barrier(n);s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow \\ (G', \{(\iota, \mathbf{run}(s, \mathit{\Gamma}'), L')\} \cup T) \end{array}}$$

$$\begin{array}{c} (G, \{(\iota, \mathbf{run}(\texttt{if true then s}_1\texttt{ else s}_2\texttt{;s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow \\ (G, \{(\iota, \mathbf{run}(\texttt{s}_1\texttt{;s}, \mathit{\Gamma}), L)\} \cup T) \end{array}$$

$$\begin{array}{c} (G, \{(\iota, \mathbf{run}(\texttt{if false then s}_1\texttt{ else s}_2\texttt{;s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow \\ (G, \{(\iota, \mathbf{run}(\texttt{s}_2\texttt{;s}, \mathit{\Gamma}), L)\} \cup T) \end{array}$$

$$\frac{\mathit{eval}(\texttt{e}, \mathit{\Gamma}) = \mathrm{b}}{\begin{array}{c} (G, \{(\iota, \mathbf{run}(\texttt{if e then s}_1\texttt{ else s}_2\texttt{;s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow \\ (G, \{(\iota, \mathbf{run}(\texttt{if } b \texttt{ then s}_1\texttt{ else s}_2\texttt{;s}, \mathit{\Gamma}), L)\} \cup T) \end{array}}$$

$$\frac{\begin{array}{c} \mathit{def}(\mathit{pn}) := \texttt{pn}(\texttt{w}_1, \ldots, \texttt{w}_\texttt{n}) \textbf{ requires } \mathit{\Phi}_{pr} \textbf{ ensures } \mathit{\Phi}_{po};\ \{\ \texttt{s}_1\ \} \\ \texttt{s}_1' = [\texttt{v}_1/\texttt{w}_1, \ldots, \texttt{v}_\texttt{n}/\texttt{w}_\texttt{n}]\texttt{s}_1 \end{array}}{\begin{array}{c} (G, \{(\iota, \mathbf{run}(\texttt{pn}(\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}); \texttt{s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow \\ (G, \{(\iota, \mathbf{run}(\texttt{s}_1'; \texttt{s}, \mathit{\Gamma}), L)\} \cup T) \end{array}}$$

$$\frac{\begin{array}{c} \mathit{def}(\mathit{pn}) := \texttt{pn}(\texttt{w}_1, \ldots, \texttt{w}_\texttt{n}) \textbf{ requires } \mathit{\Phi}_{pr} \textbf{ ensures } \mathit{\Phi}_{po};\ \{\ \texttt{s}_1\ \} \\ \forall i \in \{1, \ldots, n\} \bullet \mathit{\Gamma}(\texttt{v}_\texttt{i}) = o_i \qquad \mathit{fresh}(\iota_1) \\ \mathit{\Gamma}_1 = [\texttt{w}_1 \mapsto o_1, \ldots, \texttt{w}_\texttt{n} \mapsto o_n] \qquad \mathit{\Gamma}' = \mathit{\Gamma}[\texttt{v} \mapsto \iota_1] \\ L_1 = [(o_i, G_p(o_i)) \mid \mathit{\Gamma}(\texttt{v}_\texttt{i}) = o_i \wedge \mathit{typeof}(o_i) = \texttt{barrier}] \end{array}}{\begin{array}{c} (G, \{(\iota, \mathbf{run}(\texttt{v} = \texttt{fork}(\texttt{pn}, \texttt{v}_1, \ldots, \texttt{v}_\texttt{n}); \texttt{s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow \\ (G, \{(\iota, \mathbf{run}(\texttt{s}, \mathit{\Gamma}'), L)\} \cup \{(\iota_1, \mathbf{run}(\texttt{s}_1, \mathit{\Gamma}_1), L_1)\} \cup T) \end{array}}$$

$$\frac{\begin{array}{c} \mathit{\Gamma}(\texttt{b}) = o \qquad G(o) = (i, t, p) \qquad i{<}t{-}1 \\ G' = G[o \mapsto (i{+}1, t, p)] \qquad L' = L[o \mapsto L(o){+}1] \end{array}}{(G, \{(\iota, \mathbf{run}(\texttt{wait}(\texttt{b}); \texttt{s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow (G', \{(\iota, \mathbf{wait}(o, \texttt{s}, \mathit{\Gamma}), L')\} \cup T)}$$

$$\frac{\begin{array}{c} \mathit{\Gamma}(\texttt{b}) = o \qquad G(o) = (i, t, p) \qquad i{=}t{-}1 \\ G' = G[o \mapsto (0, t, p{+}1)] \qquad L' = L[o \mapsto L(o){+}1] \end{array}}{(G, \{(\iota, \mathbf{run}(\texttt{wait}(\texttt{b}); \texttt{s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow (G', \{(\iota, \mathbf{wait}(o, \texttt{s}, \mathit{\Gamma}), L')\} \cup T)}$$

$$\frac{L(o) = G_p(o)}{(G, \{(\iota, \mathbf{wait}(o, \texttt{s}, \mathit{\Gamma}), \mathit{\Gamma}), L)\} \cup T) \rightarrow (G, \{(\iota, \mathbf{run}(\texttt{s}, \mathit{\Gamma}), L)\} \cup T)}$$

$$\frac{\mathit{\Gamma}(\texttt{b}) = o \qquad \mathit{\Gamma}(\texttt{m}) = a \qquad a{>}0 \qquad G(o) = (i, t, p) \qquad G' = G[o \mapsto (i, t + a, p)]}{(G, \{(\iota, \mathbf{run}(\texttt{add}(\texttt{b}, \texttt{m}); \texttt{s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow (G', \{(\iota, \mathbf{run}(\texttt{s}, \mathit{\Gamma}), L)\} \cup T)}$$

$$\frac{\mathit{\Gamma}(\texttt{b}) = o \quad \mathit{\Gamma}(\texttt{m}) = a \quad G(o) = (i, t, p) \quad t{\geq}a{>}0 \quad t{-}a{>}i \quad G' = G[o \mapsto (i, t{-}a, p)]}{(G, \{(\iota, \mathbf{run}(\texttt{remove}(\texttt{b}, \texttt{m}); \texttt{s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow (G', \{(\iota, \mathbf{run}(\texttt{s}, \mathit{\Gamma}), L)\} \cup T)}$$

$$\frac{\mathit{\Gamma}(\texttt{b}) = o \quad \mathit{\Gamma}(\texttt{m}) = a \quad G(o) = (i, t, p) \quad t{\geq}a{>}0 \quad t{-}a{\leq}i \quad G' = G[o \mapsto (0, t{-}a, p{+}1)]}{(G, \{(\iota, \mathbf{run}(\texttt{remove}(\texttt{b}, \texttt{m}); \texttt{s}, \mathit{\Gamma}), L)\} \cup T) \rightarrow (G', \{(\iota, \mathbf{run}(\texttt{s}, \mathit{\Gamma}), L)\} \cup T)}$$

$$(G, \{(\iota, \mathbf{run}(\texttt{skip}, \mathit{\Gamma}), L)\} \cup T) \rightarrow (G, \{(\iota, \mathbf{done}, L)\} \cup T)$$

Fig. 14. Small-step Operational Semantics