Modular Verification of Finite Blocking in Non-terminating Programs

Modular Verification of Finite Blocking in Non-terminating Programs

Pontus Boström[1] and Peter Müller[2]

1 **Åbo Akademi University, Finland,** pontus.bostrom@abo.fi
2 **Department of Computer Science, ETH Zurich, Switzerland,** peter.mueller@inf.ethz.ch

Abstract

Most multi-threaded programs synchronize threads via blocking operations such as acquiring locks or joining other threads. An important correctness property of such programs is for each thread to make progress, that is, not to be blocked forever. For programs in which all threads terminate, progress essentially follows from deadlock freedom. However, for the common case that a program contains non-terminating threads such as servers or actors, deadlock freedom is not sufficient. For instance, a thread may be blocked forever by a non-terminating thread if it attempts to join that thread or to acquire a lock held by that thread.

In this paper, we present a verification technique for finite blocking in non-terminating programs. The key idea is to track explicitly whether a thread has an obligation to perform an operation that unblocks another thread, for instance, an obligation to release a lock or to terminate. Each obligation is associated with a measure to ensure that it is fulfilled within finitely many steps. Obligations may be used in specifications, which makes verification modular. We formalize our technique via an encoding into Boogie, which treats different kinds of obligations uniformly. It subsumes termination checking as a special case.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.1.3 Concurrent Programming

Keywords and phrases Program verification, concurrency, liveness, progress, obligations

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.639

1 Introduction

Most multi-threaded programs synchronize threads via blocking operations such as acquiring locks, receiving messages on a channel, awaiting conditions, or joining other threads. The correctness of such programs typically relies on all threads being able to make progress, that is, not being blocked forever. For instance, a producer-consumer system typically requires that each producer will eventually succeed in acquiring the lock to a shared buffer. Existing work [10, 15] has demonstrated that for *terminating* programs, progress can be ensured by (1) avoiding starvation through fair scheduling and (2) showing that the program does not create circular situations akin to deadlock, where each thread on a cycle waits for the next thread to perform an action to unblock it.

However, this solution is insufficient if programs contain potentially non-terminating threads such as actors, servers, watch-dogs, etc. Such threads potentially defer the execution of an unblocking operations forever. For instance, a thread may be blocked forever by a non-terminating thread if it attempts to join that thread or to acquire a lock held by that thread.

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 639–663

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we present a verification technique for finite blocking in non-terminating programs. The key idea is to track explicitly whether a thread has an obligation to perform an operation that unblocks another thread. For instance, a thread may receive on a channel only if another thread has an obligation to send a message on that channel, and a thread may join another thread only if the latter has an obligation to terminate. To handle non-termination, we associate each obligation with a measure (also called variant or ranking function) and check that each thread satisfies its obligations within finitely many steps, even if the thread does not terminate. Our verification technique guarantees *finite blocking* for programs with a finite number of threads in each state and fair scheduling. That is, each thread in an execution of a verified program either terminates or runs forever, but no thread is blocked forever.

Even though the finite blocking guarantee relies on fairness, our technique is also useful for non-fair systems. First, proving that no thread postpones unblocking another thread indefinitely is still necessary (although not sufficient without fairness) for progress; a violation of this property is an error. Second, although this paper focuses on finite blocking, the concept of obligations is more general and can be used to specify and verify other liveness properties for both sequential and concurrent programs, for instance, that each asynchronous task will be awaited or that a given I/O operation will be performed eventually.

Our verification technique is modular, that is, verifies each method independently, without knowledge of the program context in which it is used and the threads executing concurrently. We formalize the technique for a language without heap memory, but the style of reasoning integrates well with permission logics such as separation logic [22] and implicit dynamic frames [23], and can be automated in a similar way. In particular, our technique produces verification conditions that are amenable to automation using SMT-solvers. We have manually encoded several challenging examples and verified them successfully in Boogie [1]. These examples include producer-consumer communicating over a channel, bi-directional channels, and parallel binary tree processing; they exercise all major features of our approach.

Contributions and Outline. This paper makes the following contributions:

1. It presents the first modular verification technique for finite blocking in non-terminating programs.
2. It introduces explicit obligations with measures to uniformly specify guarantee properties [16] and verify them in standard program logics.
3. It unifies verification tasks such as proving termination, deadlock freedom, and finite blocking in one coherent methodology.
4. It adopts ideas from the Chalice verifier [15], but encodes them in a simpler way and fixes a soundness problem.

We give an informal overview of our verification technique in Sec. 2 and introduce the programming and assertion language in Sec. 3. We present the encoding of assertions in Sec. 4 and of statements in Sec. 5. Sec. 6 provides an informal soundness argument. We discuss related work in Sec. 7 and conclude in Sec. 8. App. A illustrates the treatment of message passing and deadlock freedom that we adopted from Chalice.

2 Verification Technique

This section presents the main ideas of our verification technique informally.

2.1 Obligations

An obligation is associated with a thread and specifies an action that this thread must eventually perform, either itself or by delegating it to another thread. The action could be executing a certain statement, establishing certain conditions, or reaching certain program points. Since this paper focuses on the verification of finite blocking, we use obligations to enforce actions that a thread must perform to unblock another thread. We introduce a different kind of obligation for each blocking operation. For instance, a releases-obligation indicates that a thread must release a given lock to unblock a thread possibly trying to acquire it, and a terminates-obligation indicates that a thread must terminate to unblock a thread possibly trying to join it.

The obligations for different blocking operations have different characteristics along three dimensions:

1. Some obligations can be accumulated (for instance, to express that several messages must be sent on a channel or that a re-entrant lock must be released several times), whereas others cannot (for instance, an obligation to terminate).
2. For some obligations, there is a dual concept of *credit*, which expresses the permission to execute a blocking operation. We view credits as negative obligations. In particular, creating a credit creates also the corresponding obligation. Credits are necessary for those blocking operations where the very first execution will block. For instance, if channels are initially empty then receiving on a channel requires a credit to ensure that some thread has the obligation to send a message eventually. In contrast, acquiring a lock does not require a credit because the very first acquire for each lock always succeeds; each acquire then creates a releases-obligation to ensure that subsequent acquires also succeed eventually.
3. Some obligations may be delegated to other threads (for instance, an obligation to send a message), whereas others may not (for instance, obligations to terminate or to release a lock).

Despite these different characteristics, our verification technique treats obligations uniformly. To enable modular verification, we track the obligations held by the current thread on the level of individual method executions rather than the entire thread. Obligations may be passed between different method executions when a method is called, when a method terminates, and when a method is forked in a different thread (but not upon thread-join, as we will discuss later). Which obligations get transferred is expressed in the method specifications, analogously to the transfer of access permissions in implicit dynamic frames [14, 23]. For each kind of obligation, we provide an assertion that can be used in method pre and postconditions. When a method is called (or forked), the obligations required in the method precondition are transferred from the caller to the callee; analogously, the obligations provided by the method postcondition are transferred from the method to its caller upon termination. Loops are treated analogously: we track obligations per loop iteration, and the loop invariant specifies the permissions required and provided by a loop iteration.

Proof rules ensure that each obligation is held by an active method execution (an execution on the stack of any thread) or loop invariant until it is satisfied. In particular, a *leak check* ensures that when a method execution terminates, all of its remaining obligations are transferred to the caller. Moreover, *well-formedness checks* ensure that obligations cannot be lost by sending them in a message (that might never get received) or by putting them in the postcondition of a forked method (since the forked thread may never get joined). However, leaking or losing credit is permitted.

```
method A(l: Lock)
{
  acquire l;
  call R(l);
}
```

```
method R(l: Lock)
  requires releases(l);
{
  release l;
}
```

Figure 1 An example illustrating the use and transfer of obligations. We omit specifications related to concepts introduced later, in particular, obligation measures and deadlock prevention.

Fig. 1 illustrates some of the concepts introduced so far. Method `A` acquires lock `l`, thereby obtaining an obligation to release it eventually. Method `R` requires a releases-obligation to `l` in its precondition. Therefore, when `A` calls `R`, its obligation is transferred to `R`. After the call, `A` does not contain any obligations and, thus, passes the leak check. Method `R` gets rid of its obligation by releasing `l` and, thus, also passes the leak check.

2.2 Obligation Measures

Obligations allow one to track modularly which method execution is expected to perform a given unblocking operation. However, the proof rules sketched above are not sufficient to prevent a non-terminating thread from blocking another thread forever. Assume method `R` from Fig. 1 was implemented as follows:

```
method R(l: Lock)
  requires releases(l);
{
  while(true)
    invariant releases(l);
  { }
  release l;
}
```

This implementation passes the leak check since no obligations are held at the end of the method or at the end of a loop iteration after the releases-obligation has been transferred to the next loop iteration. However, the method obviously fails to release `l` because it enters a non-terminating loop before reaching the release operation.

A naïve solution would be to require that a method holds no obligations when it enters a possibly non-terminating loop or calls a possibly non-terminating method. However, this solution is too restrictive for many useful implementations. For instance, the `Await` method in Fig. 2 encodes a busy version of Java's `wait` method. The method loops until a condition `P` holds, where `P` refers to fields that are protected by a lock `l`. Therefore, `Await` will be called in states where the executing thread holds lock `l` and, hence, the method has a releases-obligation for `l`. In each loop iteration, the method releases and then re-acquires the look such that other threads may obtain the lock and establish `P`.

The naïve solution would disallow method `Await` unless one could prove that the loop will always terminate, which may be difficult in a modular setting. However, since the loop releases and re-acquires the lock `l` in each iteration, it is guaranteed not to block indefinitely any other thread that attempts to acquire `l` (assuming fair scheduling and fair locks). A similar situation occurs when a thread is expected to send an unbounded number of messages over a channel. Its send-loop might not be guaranteed to terminate, but holds a sends-obligation in each iteration (see App. A for the full example); it would therefore be rejected by the naïve solution.

```
method Await(l: Lock)
  requires releases(l, 1);
  ensures releases(l, 1);
{
  while(!P)
    invariant releases(l, 1);
  {
    release l;
    acquire l;
  }
}
```

Figure 2 A busy version of Java's `wait` method. In contrast to method `R` above, the lock is released and re-acquired in each loop iteration. We omit specifications related to deadlock prevention.

Measures. These two examples show that the naïve solution is overly conservative. It should be possible for a thread to hold obligations during a non-terminating execution as long as these obligations will be satisfied eventually. To verify this liveness property without resorting to temporal reasoning, we reduce it to a safety property by associating each obligation with a measure (also called variant or ranking function). Analogously to a termination measure, an obligation's measure is an expression that evaluates to a value in a well-founded set. Proof rules ensure that the measure is decreased in each loop iteration or recursive call, and that the obligation gets satisfied before its measure expires. This check would fail for the non-terminating version of method `R` above because there is no measure that one could choose for the releases-obligation that gets decreased during the non-terminating loop.

Fresh Obligations. Measures alone cannot distinguish between the situations in method `R` and method `Await`. In both of them, a possibly non-terminating loop holds a releases-obligation before and after each loop iteration. However, method `R` might cause indefinite blocking because the obligation is held throughout the loop body, whereas method `Await` is safe because the releases-obligation is satisfied and re-obtained in each iteration, giving other threads a chance to acquire the lock in between. To distinguish these two situations, we track explicitly whether an obligation is *fresh*, that is, has been obtained since the prestate of the current method execution or loop iteration. Fresh obligations are exempted from the check that their measure decreases before the next recursive call or loop iteration. In the `Await` method above, the measure of all releases-obligations is the constant 1, expressing that lock `l` will be released within one loop iteration. This constant measure is not decreased in the loop body. However, since acquiring the lock `l` obtains a fresh releases-obligation, it is exempted from the check that the measure decreases, and the method verifies.

Termination. Associating obligations with measures allows us to treat termination like any other obligation. Therefore, termination proofs are a special case of the general technique we propose. For instance, the factorial method in Fig. 3 promises to terminate after at most `n` recursive calls if its argument is non-negative. This termination guarantee is expressed by including a terminates-obligation with measure `n` in the method's precondition. We assume here that programmers provide the measures for termination and other obligations. Combining our technique with inference of termination measures (see Cook et al. [4] for an overview) is future work. The termination guarantee of `Fac` ensures that the join in method `Main` will not block indefinitely.

It might initially seem un-intuitive that termination as well as the satisfaction of other

```
method Fac(n: int) returns (res: int)
  requires 0 ≤ n ⇒ terminates(n);
{
  if (n ≤ 1)
    res := 1;
  else
    res := n * Fac(n - 1);
}
```

```
method Main(n: int) returns (res: int)

{
  if (0 ≤ n) {
    fork t := Fac(n);
    join res := t;
  }
}
```

Figure 3 A recursive factorial method. The terminates-obligation in the precondition expresses that the method will terminate if `n` is non-negative. The antecendent ensures that the measure of the obligation is well-founded. The main method forks a new thread to execute `Fac`. It may join this thread only because `Fac` is guaranteed to terminate and, thus, the join will not block indefinitely.

obligations is specified as a method precondition rather than a postcondition. However, this approach is consistent with the treatment of permissions in permission-based logics such as separation logic. The precondition specifies which resources get transferred from the caller to the callee. In permission logics, the transferred resources are partial heaps; here, they are obligations. So one should think of a precondition as the obligations *consumed* by the callee and of a postcondition as the obligations *provided* by the callee.

2.3 Wait Order

Finite blocking implies the absence of deadlock because each thread involved in a deadlock blocks indefinitely. A deadlock occurs if one or more threads form a cycle where each thread is blocked by its successor on the cycle. Obligations allow us to define this blocked-by relation uniformly for different blocking operations: a thread t is blocked by another thread t' if t is blocked on a blocking operation and t' holds an obligation to unblock it. For instance, t is blocked by t' if t tries to acquire a lock and t' holds the lock (and thus has an obligation to release it), or if t tries to join t' (and thus t' has an obligation to terminate).

We guarantee deadlock freedom by preventing cycles in the blocked-by relation. For this purpose, we introduce a strict partial order on threads and ensure via proof obligations for all blocking operations that a thread t may be blocked by a thread t' only if t is (strictly) less than t'. The order on threads is defined by letting the programmer define a strict partial *wait order* on obligations. A thread t is less than t' if for each obligation o held by t there exists an obligation o' held by t' such that o is less than o'. Cycles in the blocked-by relation are then prevented by proving for each blocking operation that each obligation held by the thread executing the blocking operation is less than the obligation to unblock it. Since this proof obligation refers only to the current thread, it can be checked in thread-modularly (we will discuss later how to check it procedure-modularly).

The wait order on obligations generalizes our earlier work [15] to arbitrary obligations. Like that work, we assume that the wait order on obligations is fixed throughout the execution of a program (but the order on threads changes dynamically as they obtain and lose obligations).

3 Programming and Assertion Language

In this section, we introduce the programming and assertion language. Their semantics will be defined in the next two sections.

$$
\begin{array}{rcl}
S & ::= & v := \texttt{new lock} \\
& | & \texttt{acquire}\ e \\
& | & \texttt{release}\ e \\
& | & v := \texttt{new}\ C \\
& | & \texttt{send}\ e_1(e_2) \\
& | & \texttt{receive}\ v := e \\
& | & \texttt{call}\ v := e_1.M(e_2) \\
& | & \texttt{fork}\ v := e_1.M(e_2) \\
& | & \texttt{join}\ v := e \\
& | & \texttt{while}(e)\ \texttt{invariant}\ A\ \{\ S_1\ \}
\end{array}
$$

Figure 4 The relevant statements of our programming language. We omitted assignment, sequential composition, and conditional statements because their treatment is straightforward. A fork statement yields a token, which can be used to join the forked thread.

3.1 Programming Language

We present our technique for a simple imperative programming language with iteration and recursion, threads, as well as dynamically-created locks and channels. For simplicity, we omit other heap-allocated objects because their treatment is orthogonal to the focus of this paper. However, our technique is compatible with permission-based program logics that handle them.

A program consists of a sequence of method declarations and channel type declarations. A method declaration has the form

```
method M(p: T1) returns (r: T2)
  requires A1;
  ensures A2;
{ S }
```

where M is a unique method name and each T_i is one of the following types: `bool`, `int`, `lock`, `token`, or a channel type. A_i are assertions and S is a statement, see below. Like in the Chalice language [15], a channel type declaration has the form

```
channel C(p: T) where A;
```

where C is a unique channel type name. Messages sent over such a channel are values of type T. The `where` clause specifies a *channel invariant*, that is, constraints on the messages; it also specifies the credits sent with each message.

Statements (Fig. 4) include operations on non-reentrant locks (creation, acquire, release), operations on channels (creation, send, receive), method call, thread fork and join, and loops with loop invariants. We also assume to have assignments, sequential composition, and conditional statements, but do not formalize them because they are straightforward. Expressions e include constants, variables v, and the usual boolean and arithmetic operations. We will explain and formalize the semantics of statements in Sec. 5.

For simplicity, channels have unbounded buffers such that send operations never block. Therefore, the blocking operations in our language are acquiring a lock, receiving a message, and joining a thread.

$$
\begin{array}{rcl}
A & ::= & e \\
& | & A_1 \texttt{ \&\& } A_2 \\
& | & e \Rightarrow A_1 \\
& | & \texttt{releases}(e_1, e_2) \\
& | & \texttt{sends}(e_1, e_2, e_3) \\
& | & \texttt{terminates}(e) \\
& | & \texttt{joinable}(e) \\
& | & \texttt{waitlevel} \ll e
\end{array}
$$

Figure 5 The assertion language. The three kinds of obligations exhibit all different characteristics of obligations discussed in Sec. 2.1.

3.2 Assertion Language

Assertions are used as method pre and postconditions, loop invariants, and channel invariants. Besides the usual constraints on variables, they specify which obligations and credits get transferred between method executions and loop iterations, along with their measures.

Measures. In order to define measures for obligations, we adopt Dafny's approach [13] and assume a pre-defined well-founded strict partial order $\sqsubset$ on all values of a program execution. For instance, for integers x and y, we define $x \sqsubset y \Leftrightarrow x < y \wedge 0 \leq y$, whereas for an integer x and a lock l, $x \sqsubset l$ is undefined. The resulting well-founded set forms a complete lattice $(\mathbb{V}, \sqsubset)$ with top element $\top$ and bottom element $\bot$. Assuming a pre-defined order simplifies the presentation of the verification technique. An adaptation to user-defined orders is possible, but reveals nothing interesting.

Wait levels. As explained in Sec. 2.3, we use a strict partial order on obligations to prove deadlock freedom. To encode this order, we assign every obligation a *wait level*, that is, a value in a dense lattice $(\mathbb{L}, \ll)$ with strict order $\ll$ and bottom element $\bot$.

Assertions. The assertion language is summarized in Fig. 5. It includes boolean expressions, conjunction, and implication. Moreover, there are assertions for three kinds of obligations. For a releases-obligation $\texttt{releases}(e_1, e_2)$, e_1 of type `lock` denotes the lock that must be released and $e_2 \in \mathbb{V}$ is the measure. For a sends-obligation $\texttt{sends}(e_1, e_2, e_3)$, e_1 is of a channel type and denotes the channel on which messages must be sent, e_2 is an integer that denotes how many messages must be sent, and e_3 is the measure. When e_2 is negative, the assertion denotes credits, that is, permissions to receive rather than obligations to send. For a terminates-obligation $\texttt{terminates}(e)$, e is the measure. For all three obligation assertions, the measure can be any value in $\mathbb{V}$, including $\top$ and $\bot$. The assertion $\texttt{joinable}(e)$, where e is of type `token` provides the permission to join the thread denoted by e. A thread may have a join-permission for e if the thread represented by the token e is guaranteed to terminate and has not been joined yet, and if no other thread has the permission to join it. The assertion $\texttt{waitlevel} \ll e$ expresses that the wait level of each obligation held by the current thread is strictly less than the wait level of e. Thus, the current thread may execute a blocking operation, where the corresponding obligation to unblock has level e, without creating a deadlock. We say that e *is above the current wait level if* $\texttt{waitlevel} \ll e$. One can think of

`waitlevel` as the maximum wait level of all obligations held by the current thread; however, we will use it to specify and check only upper bounds one these levels.

Conjunction `&&` is analogous to separating conjunction in separation logic. In particular, `releases`(l, n) `&&` `releases`(l, n) expresses that the current thread must release lock l twice. Since this is not possible for non-reentrant locks, the conjunction is equivalent to false. The conjunction `sends`$(c, 1, n)$ `&&` `sends`$(c, 1, n)$ expresses that the current thread has *two* obligations to send a message on channel c; that is, it is equivalent to `sends`$(c, 2, n)$.

Note that the use of sends-obligations and credits is not new [15] (see App. A for an example). We include them here to demonstrate how our technique handles a range of obligations uniformly and to exhibit all different characteristics of obligations discussed in Sec. 2.1. Sends-obligations can be accumulated, have the dual concept of sends-credits, and can be transferred between threads, whereas releases-obligations and terminates-obligations cannot be accumulated, have no credits, and cannot be transferred. Therefore, our assertion language is representative for a wide range of obligations including for instance obligations to await an asynchronous task or perform I/O.

Well-formedness Conditions. We impose several well-formedness conditions on assertions.

1. Method postconditions must not contain terminates-obligations because these obligations are satisfied when the method terminates and, thus, not returned to the caller.
2. A method may be forked only if its precondition does not contain any releases-obligations. This condition reflects that a lock must be released by the thread that acquired it; neither the held lock nor the releases-obligation can be transferred to another thread.
3. A method may be forked only if its postcondition does not contain any obligations. This condition prevents leaking of obligations when a forked thread is never joined. For terminates and releases-obligations, this condition can be checked syntactically. If the postcondition contains an assertion `sends`(c, e, n), we verify that e evaluates to a non-positive number, that is, the assertion denotes a credit.
4. A channel invariant must not contain any obligations (but credits are allowed). This condition ensures that obligations cannot be leaked by sending them in a message that might never get received.
5. A channel invariant must not contain wait level constraints because these constraints cannot be interpreted consistently in the sending and receiving thread of a message.

4 Encoding of Assertions

In this section, we present an encoding of assertions into a guarded command language similar to Boogie [1]. For readability, we use dedicated operators and constant symbols for measures and wait levels rather than Boogie's uninterpreted functions, and bulk updates (**foreach** statements) instead of encoding them via Boogie's `havoc` and `assume` statements. In the following, we introduce the representation of program states, explain how we encode the transfer of obligations, and then formalize the meaning of assertions.

4.1 Encoding of States

The state of a method execution consists of the method's parameter and result variables, its local variables, as well as the obligations (and credits) held by this method execution. To treat the different kinds of obligations uniformly, we introduce a type

$$\mathbf{obl} = \mathbf{lock} \cup \mathbf{channel} \cup \{\mathbf{term}\}$$

where **channel** includes all channel types declared in the program. Here, a lock identifies a releases-obligation, a channel identifies a sends-obligation (or credit), and the identifier **term** identifies a terminates-obligation. Using the **obl** type, we declare a global map that stores the obligations and credits held by the current method execution or loop iteration:

$$\mathcal{B} : \mathbf{obl} \to \mathbb{Z}$$

$\mathcal{B}[o] = n$ encodes that the current method execution has n obligations for o if n is positive, and $-n$ credits if n is negative, The latter occurs only if o is a channel.

As we explained in Sec. 2.2, we track separately which obligations are fresh, that is, have been obtained since the prestate of the current method execution or loop iteration. The number of fresh obligations is stored in a global map:

$$\mathcal{F} : \mathbf{obl} \to \mathbb{N}$$

$\mathcal{F}[o]$ yields how many of the obligations in $\mathcal{B}[o]$ are fresh. If there are no obligations, $\mathcal{F}[o]$ is zero. That is, the following invariants hold in all states:

$$\forall o \in \mathbf{obl} \cdot 0 \leq \mathcal{B}[o] \Rightarrow \mathcal{F}[o] \leq \mathcal{B}[o]$$
$$\forall o \in \mathbf{obl} \cdot \mathcal{B}[o] \leq 0 \Rightarrow \mathcal{F}[o] = 0$$

Since the first execution of a join statement for any thread t may block, we need in principle a credit that provides the permission to join t (see the characteristics of obligations in Sec. 2.1). This credit is the dual of the terminates-obligation for t. That is, the forker of t obtains the credit needed to join t if t promises to terminate, that is, consumes a terminates-obligation. It is possible to encode join-permissions as terminates-credits, but such an encoding complicates `terminates` assertions (which would need an argument that identifies the thread) and the encoding of fork (since terminates-obligations in the precondition of the forked method must be interpreted differently in the forker and in the forkee). Therefore, we choose a different encoding here. The map $\mathcal{B}$ does not contain termination information about threads other than the current thread; such information is stored in a separate map that yields whether a thread may be joined:

$$\mathcal{J} : \mathbf{token} \to \mathbb{B}$$

Finally, we record the wait level of each obligation in the following map, where $\mathbb{L}$ is the set of wait levels:

$$\mathcal{L} : \mathbf{obl} \cup \mathbf{token} \to \mathbb{L}$$

For a lock or channel o, $\mathcal{L}[o]$ denotes the wait level of the corresponding releases- or sends-obligations. $\mathcal{L}[\mathbf{term}]$ denotes the level of the terminates-obligation of the current thread, and for a token t, $\mathcal{L}[t]$ denotes the level of the terminates-obligation of the thread represented by t.

4.2 Transfer of Obligations and Credits

Our assertions do not only express conditions on the state but also specify which obligations (and credits) get transferred between method executions and loop iterations. This behavior is similar to assertions in permission logics, which describe how ownership of resources is transferred. We formalize the meaning of assertions via two operations, exhale and inhale (sometimes called produce and consume). In this subsection, we explain how to exhale and inhale obligations and credits. A key virtue of our approach is that these operations are uniform for all kinds of obligations. Exhaling and inhaling assertions will be explained in the next subsection.

$$
\begin{array}{l}
\mathit{Exhale}_{obl}(o, n, m, \mathit{creditsAllowed}, P) = \\
\quad \textbf{assert } \mathit{creditsAllowed} \vee n \leq \mathcal{B}[o]; \\
\quad \textbf{if } (m = \top) \ \{ \\
\quad\quad \textbf{assert } n \leq \mathcal{F}[o] \vee \mathcal{B}[o] \leq \mathcal{F}[o]; \\
\quad\quad \mathcal{F}[o] := \mathit{max}(\mathcal{F}[o] - n, 0); \\
\quad \} \textbf{ else } \ \{ \\
\quad\quad \textbf{assert } 0 < n \wedge \mathcal{F}[o] < \mathcal{B}[o] \Rightarrow m \sqsubset P[o]; \\
\quad \} \\
\quad \mathcal{B}[o] := \mathcal{B}[o] - n; \\
\quad \textbf{if } (\mathcal{B}[o] < \mathcal{F}[o]) \ \{ \\
\quad\quad \mathcal{F}[o] := \mathit{max}(\mathcal{B}[o], 0); \\
\quad \}
\end{array}
$$

Figure 6 The exhale operation for obligations and credits. $o \in \textbf{obl}$ is the obligation, $n \in \mathbb{Z}$ indicates the number of obligations (or credits) to exhale, and $m \in \mathbb{V}$ is the measure of the obligations to be exhaled. The boolean flag *creditsAllowed* indicates whether credits are allowed for the kind of obligations to be exhaled. $P \in \textbf{obl} \to \mathbb{V}$ provides the measure of obligations in the prestate of the enclosing method or loop for the check that the measure decreases.

Exhale. Exhaling obligations is formalized in Fig. 6. $\mathit{Exhale}_{obl}(o, n, m, \mathit{creditsAllowed}, P)$ exhales n obligations (or $-n$ credits, if n is negative) for o (where o is a lock, channel, or **term**) with measure m. It first asserts that credits are permitted for this kind of obligation or that the current state has enough obligations to exhale. (Applications of Exhale_{obl} will ensure that *creditsAllowed* is true if n is negative.)

For the rest of the operation, let us first consider the case that we exhale obligations, that is, $0 < n$. If the exhaled obligations are fresh (indicated by $m = \top$), we check that there are enough fresh obligations available or that there are no non-fresh obligations. In the former case, the fresh obligations are given away. In the latter case, the exhale gives away all available fresh obligations and obtains some credits. It would be unsound to exhale fresh obligations if neither case applied because reducing the number of obligations would then treat non-fresh obligations as fresh, thereby providing a way to postpone their satisfaction. If the exhaled obligations are non-fresh ($m \neq \top$), we check that if the current state holds non-fresh obligations ($\mathcal{F}[o] < \mathcal{B}[o]$), their measure decreased w.r.t. the prestate measure of the enclosing method or loop, provided by the map $P \in \textbf{obl} \to \mathbb{V}$. In both cases, we remove the exhaled obligations from the state and adjust the number of fresh obligations to maintain the invariants mentioned in Sec. 4.1.

If we exhale credits (that is, $n \leq 0$), the assertions in both branches of the conditional hold trivially (recall that $0 \leq \mathcal{F}[o]$). Giving away fresh credits increases the number of fresh obligations, and giving away any credits always increases the number of obligations. It is therefore preferable to make all credits in assertions fresh.

Creation and Cancellation of Credits. A sends-credit for a channel c is created by exhaling a sends-obligation for c in a state that holds no such obligation. However, the inverse operation—canceling an obligation with a credit— is *not* permitted. That is, inhaling a credit in a state that holds a corresponding obligation, or inhaling an obligation in a state that holds a corresponding credit leads to a verification error. It would be unsound to create a credit by exhaling an obligation with a small measure and then cancel the credit with

$$
\begin{array}{l}
\mathit{Inhale}_{obl}(o, n, m, P) = \\
\quad \textbf{if } (0 < n) \ \{ \\
\quad\quad P[o] := P[o] \sqcap m; \ \} \\
\quad \} \\
\quad \textbf{assert } (0 < n \Rightarrow 0 \leq \mathcal{B}[o]) \wedge (n < 0 \Rightarrow \mathcal{B}[o] \leq 0); \\
\quad \mathit{Exhale}_{obl}(o, -n, m, \texttt{true}, P_\top);
\end{array}
$$

Figure 7 The inhale operation for obligations and credits. The parameters o, n, and m are analogous to Exhale_{obl}. Inhaling obligations records their measures in the map $P \in \textbf{obl} \to \mathbb{V}$ for later checks. Note that we treat the inhale operation as a parameterized macro such that updates to P modify the argument map at the call site. $P_\top \in \textbf{obl} \to \mathbb{V}$ yields $\top$ for all obligations and is used to suppress the measure check in Exhale_{obl}, which is not needed during inhale.

an obligation that has a larger measure. This would effectively increase the measure of the obligation and, thus, provide a way to postpone the satisfaction of the obligation indefinitely. Even if the obligations involved in creating and canceling a credit had the same measure, one could postpone the satisfaction of the obligation indefinitely by arranging a sequence of threads where each thread obtains a credit from its successor to cancel its own obligation, creating another obligation in the successor, and so on.

One could prevent this unsoundness by recording the measure of the exhaled obligation when creating a credit and then enforcing that the credit may cancel only obligations that have a strictly larger measure. Since this solution requires substantial bookkeeping and since the main purpose of sends-credits is to enable receive operations (rather than canceling sends-obligations), we simply forbid cancellation of obligations and credits altogether. This rule is reflected in the encoding of inhale below.

Inhale. Inhaling obligations is formalized in Fig. 7. $\mathit{Inhale}_{obl}(o, n, m, P)$ inhales n obligations (or $-n$ credits, if n is negative) for o with measure m. The operation records the measures of inhaled obligations in map P. If there are multiple obligations for o, we abstract their measures by storing their minimum. This is achieved by using the meet $\sqcap$ of the measure lattice. We treat the inhale operation as a parameterized macro such that updates to P modify the argument map at the call site (that is, P behaves like an in-out parameter). We will record measures only in the prestates of method executions and loop iterations; in all other cases, we will pass a dummy map for P.

The assertion after the update of P prevents credit cancellation as explained above. Finally, obligations are added by exhaling the corresponding credits, and vice versa. Since the decrease-checks for measures before recursive calls and at the end of loop iterations will be encoded via exhale, inhale does not have to perform any such checks. Therefore, it passes $P_\top$, which yields $\top$ for all obligations, to Exhale_{obl}, such that the check $m \sqsubset P[o]$ there will trivially succeed.

4.3 Exhaling and Inhaling Assertions

Exhaling an assertion A checks that the constraints specified by A hold and removes the obligations and credits specified in A from the current state. The definition is provided in Fig. 8. Exhaling proceeds in two phases. The first phase checks all constraints except those on wait level and handles the transfer of obligations and credits. The second phase only checks

$$
\begin{array}{lcl}
\mathit{Exhale}(A, P) & = & \mathit{Exhale}_1(A, P)\ \mathit{Exhale}_2(A, \_) \\
\\
\mathit{Exhale}_i(A_1\ \texttt{\&\&}\ A_2, P) & = & \mathit{Exhale}_i(A_1, P)\ \mathit{Exhale}_i(A_2, P) \\
\mathit{Exhale}_i(e \Rightarrow A, P) & = & \textbf{if}\ (\llbracket e \rrbracket)\ \{\ \mathit{Exhale}_i(A, P)\ \} \\
\\
\mathit{Exhale}_1(e, \_) & = & \textbf{assert}\ \llbracket e \rrbracket; \\
\mathit{Exhale}_1(\texttt{releases}(e_1, e_2), P) & = & \mathit{Exhale}_{obl}(\llbracket e_1 \rrbracket, 1, \llbracket e_2 \rrbracket, \texttt{false}, P) \\
\mathit{Exhale}_1(\texttt{sends}(e_1, e_2, e_3), P) & = & \mathit{Exhale}_{obl}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, \llbracket e_3 \rrbracket, \texttt{true}, P) \\
\mathit{Exhale}_1(\texttt{terminates}(e), P) & = & \mathit{Exhale}_{obl}(\textbf{term}, 1, \llbracket e \rrbracket, \texttt{true}, P) \\
\mathit{Exhale}_1(\texttt{joinable}(e), \_) & = & \textbf{assert}\ \mathcal{J}[\llbracket e \rrbracket]; \mathcal{J}[\llbracket e \rrbracket] := \texttt{false}; \\
\\
\mathit{Exhale}_2(\texttt{waitlevel} \ll e, \_) & = & \textbf{assert}\ \mathit{levelBelow}(\mathcal{B}, \mathcal{L}[\llbracket e \rrbracket]);
\end{array}
$$

Figure 8 Encoding of exhale. A is an assertion, and $P \in \textbf{obl} \to \mathbb{V}$ provides the prestate measures for the check that obligation measures decrease. All cases not mentioned here are defined as **skip**. $\llbracket \_ \rrbracket$ encodes expressions of the programming language; it is straightforward and, therefore, omitted.

wait level constraints. This encoding via two phases is necessary to treat wait level constraints soundly. It checks wait level constraints during exhale *after* obligations and credits have been removed from the state, and assumes wait level constraints during inhale *before* obligations and credits have been added (see Fig. 9 below). That is, in both cases, `waitlevel` refers to a state that does not contain the transferred obligations and credits. This fixes an unsoundness in Chalice [15], where it was possible to interpret `waitlevel` inconsistently during exhale and inhale and, thus, exhale assertions that lead to an inconsistency when inhaled.

In both phases of exhale, conjunction is treated multiplicatively by sequentially exhaling the two conjuncts. This is analogous to an encoding of separating conjunction [21]. Implication is encoded via a conditional statement.

Phase 1 uses Exhale_{obl} from Fig. 6 to transfer obligations and credits, and to check that measures decrease. Even though there are no terminates-credits, we set the *creditsAllowed* parameter of Exhale_{obl} to true for terminates-obligations because our encoding of statements will lead to intermediate states with a negative number of terminates-obligations. Exhaling a join-permission asserts that such a permission is held and removes it.

Phase 2 checks wait level constraints. In order to be useful to prove deadlock freedom, $\texttt{waitlevel} \ll e$ expresses that the wait level of each obligation *held by the current thread* is strictly less than the wait level of e. In our procedure-modular verification technique, we cannot check this condition directly because we record (in map $\mathcal{B}$) only the obligations *held by the current method execution or loop iteration*, but not those held by other method executions on the call stack or enclosing loops. To account for those, our encoding uses a local variable *residue* $\in$ **token** in each method. We leave the value of *residue* unspecified, but ensure that we can prove that its level is less than an upper bound u only if the level of all obligations held by the current thread, but not by the current method execution or loop iteration, is less than u. Therefore, we can prove $\texttt{waitlevel} \ll e$ by proving that the level of all obligations recorded in $\mathcal{B}$ as well as the level of *residue* are less than e's level. We encode this via the following predicate:

$$\mathit{levelBelow}(B, u) = (\forall o \in \textbf{obl} \cdot 0 < B[o] \Rightarrow \mathcal{L}[o] \ll u) \wedge \mathcal{L}[\mathit{residue}] \ll u$$

Our encoding ensures that the only information obtained about *residue*'s level is the upper

$$
\begin{array}{lcl}
\mathit{Inhale}(A, P) & = & \texttt{var}\ \mathcal{B}_{old} := \mathcal{B}; \mathit{Inhale}_1(A, P) \\
\\
\mathit{Inhale}_1(A_1\ \texttt{\&\&}\ A_2, P) & = & \mathit{Inhale}_1(A_1, P)\ \mathit{Inhale}_1(A_2, P) \\
\mathit{Inhale}_1(e \Rightarrow A, P) & = & \textbf{if}\ (\llbracket e \rrbracket)\ \{\ \mathit{Inhale}_1(A, P)\ \} \\
\mathit{Inhale}_1(e, \_) & = & \textbf{assume}\ \llbracket e \rrbracket; \\
\mathit{Inhale}_1(\texttt{releases}(e_1, e_2), P) & = & \mathit{Inhale}_{obl}(\llbracket e_1 \rrbracket, 1, \llbracket e_2 \rrbracket, P) \\
\mathit{Inhale}_1(\texttt{sends}(e_1, e_2, e_3), P) & = & \mathit{Inhale}_{obl}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, \llbracket e_3 \rrbracket, P) \\
\mathit{Inhale}_1(\texttt{terminates}(e), P) & = & \mathit{Inhale}_{obl}(\textbf{term}, 1, \llbracket e \rrbracket, P) \\
\mathit{Inhale}_1(\texttt{joinable}(e), \_) & = & \mathcal{J}[\llbracket e \rrbracket] := \texttt{true}; \\
\mathit{Inhale}_1(\texttt{waitlevel} \ll e, \_) & = & \textbf{assume}\ \mathit{levelBelow}(\mathcal{B}_{old}, \mathcal{L}[\llbracket e \rrbracket]);
\end{array}
$$

Figure 9 Encoding of inhale. A is an assertion, and $P \in \textbf{obl} \to \mathbb{V}$ is used to record the measures of inhaled obligations.

bounds when inhaling wait level constraints as part of method pre or postconditions, or loop invariants. Therefore, the prover needs check assertions for any value of *residue*'s level below these upper bounds, including a value above the levels of the obligations held by the transitive callers of the current method. To understand why such a value always exists, consider a method m with precondition $\texttt{waitlevel} \ll e$. This condition constrains the level of m's *residue* variable to be less than e's level. When exhaling this precondition in the caller n, we check that the levels of all obligations held by n are less than e's level. Therefore, since wait levels form a dense lattice, there exists a possible value for the level of m's *residue* variable that is above all obligations held by n and less than e's level. By checking (as part of exhaling the precondition) that the level of n's *residue* variable is less than e's level, we know that there exists a value for the level of m's *residue* variable that is above the level of n's *residue* variable and less than e's level. The argument applies inductively to n's *residue* variable, the one in n's caller, and so on. That is, m's *residue* also reflects the obligations held by those method executions. The argument is analogous for enclosing loops.

The definition of inhale in Fig. 9 is analogous to exhale. It stores the current obligations map $\mathcal{B}$ before transferring obligations or credits in order to interpret wait level constraints consistently with exhale. Inhaling a constraint assumes it. Obligations and credits are transferred using the Inhale_{obl} macro from Fig. 7, and join-permissions are inhaled by adding them.

5 Encoding of Methods and Statements

In this section, we present the proof rules for our verification technique via an encoding into Boogie [1]. The resulting Boogie program contains neither obligations (which are encoded by accesses to the maps $\mathcal{B}$ and $\mathcal{F}$) nor exhale and inhale operations (which are replaced by their definitions). Therefore, we can verify the program by computing weakest preconditions over the guarded commands and proving them in an SMT solver. Verification is procedure and thread-modular. That is, each method is verified without considering its caller or interference from other threads.

$\langle\!\langle$ **method** $M(p)$ **returns** (r) **requires** $pre_M(\mathbf{this}, p)$ **ensures** $post_M(\mathbf{this}, p, r)$ { S } $\rangle\!\rangle =$
  assume $\forall o \in \mathbf{obl} \cdot \mathcal{B}[o] = 0$;
  var *residue*;
  var $P_{method} := P_\top$;
  $Inhale(pre_M(\mathbf{this}, p), P_{method})$
  foreach $o \in \mathbf{obl}$ { $\mathcal{F}[o] := 0$; }
  $[\![S]\!]$
  $Exhale(post_M(\mathbf{this}, p, r), P_\top)$
  $\mathcal{B}[\mathbf{term}] := 0$;
  assert $\forall o \in \mathbf{obl} \cdot \mathcal{B}[o] \leq 0$;

Figure 10 Encoding $\langle\!\langle \_ \rangle\!\rangle$ of methods. The assertions $pre_M(\mathbf{this}, p)$ and $post_M(\mathbf{this}, p, r)$ are the method precondition and postcondition, resp.

5.1 Methods

Fig. 10 shows the encoding of methods. Before inhaling the precondition, the execution of a method holds neither obligations nor credits. The value of the local variable *residue* is unspecified; its level is constrained when inhaling wait level constraints. The subsequent inhale operation assumes the method precondition and transfers obligations and credits from the caller to the callee. It records the measures of obligations in a map P_{method}, which will be used in call and fork statements to ensure that measures decrease. The recording works by passing the all-top map $P_\top$ into the inhale macro, which, for each inhaled obligation, takes the minimum (that is, the meet) of the stored measure and the measure of the inhaled obligation (see Fig. 7). After inhaling the precondition, we make all fresh obligations non-fresh since obligations that are fresh to the caller are not fresh to the callee as they existed before the execution of the callee started. This step is necessary to prevent fresh obligations from being transferred indefinitely from method execution to method execution.

The method body is encoded using the encoding function for statements $[\![\_]\!]$. After executing the body, we exhale the postcondition. During this exhale, we do not need to check that measures have decreased (which happens only at call and fork sites and at the end of loop iterations). Therefore, we pass the all-top map $P_\top$ as last argument to the exhale operation such that the decrease-check succeeds trivially. After the exhale, we remove the terminates-obligation from the obligation map since the method is about to terminate. The final step is the leak check: upon termination, the method may hold no obligations. That is, all obligations passed in from the caller or obtained during the execution of the method must be satisfied, transferred to other threads (during a fork), or returned to the caller when exhaling the postcondition.

5.2 Call, Fork, and Join

A call statement (Fig. 11) is verified by exhaling the precondition of the callee and then inhaling its postcondition. The exhale needs to check that the measures of exhaled obligations decreased since the prestate of the caller. This is achieved by passing the measures from this state (variable P_{method}, which is initialized at the beginning of the enclosing method, see Fig. 10) into the exhale operation. After the exhale, we assert that the caller retains no obligations unless the callee promises to terminate. This assertion ensures obligations cannot be left behind in the caller in cases where the control flow might never return. The

$$
\begin{array}{lcl}
[\![\texttt{call } v := e_1.M(e_2)]\!] & = & \textbf{var } \mathit{term} := \mathcal{B}[\textbf{term}]; \\
& & \mathit{Exhale}(\mathit{pre}_M([\![e_1]\!], [\![e_2]\!]), P_{\mathit{method}}) \\
& & \textbf{assert } \forall o \in \textbf{obl} \cdot \mathcal{B}[o] \leq 0 \vee \mathcal{B}[\textbf{term}] < \mathit{term}; \\
& & \mathcal{B}[\textbf{term}] := \mathit{term}; \\
& & \mathit{Inhale}(\mathit{post}_M([\![e_1]\!], [\![e_2]\!], v), P_d) \\
\\
[\![\texttt{fork } v := e_1.M(e_2)]\!] & = & \textbf{var } \mathit{term} := \mathcal{B}[\textbf{term}]; \\
& & \mathit{Exhale}(\mathit{pre}_M([\![e_1]\!], [\![e_2]\!]), P_{\mathit{method}}) \\
& & \textbf{havoc } v; \textbf{assume } \mathcal{L}[v] = \bot; \\
& & \mathcal{J}[v] := (\mathcal{B}[\textbf{term}] < \mathit{term}); \\
& & \mathcal{B}[\textbf{term}] := \mathit{term}; \\
& & \textbf{havoc } w; \textbf{assume } \mathit{levelBelow}(\mathcal{B}, w); \\
& & \mathcal{L}[v] := w; \\
\\
[\![\texttt{join } v := e]\!] & = & \textbf{assert } \mathit{levelBelow}(\mathcal{B}, [\![e]\!]); \\
& & \textbf{assert } \mathcal{J}[[\![e]\!]]; \\
& & \mathit{Inhale}(\mathit{post}_e(v), P_d) \\
& & \mathcal{J}[[\![e]\!]] := \texttt{false};
\end{array}
$$

Figure 11 Encoding of call, fork, and join statements. $P_d \in \textbf{obl} \to \mathbb{V}$ is a dummy map that is never read. The function post_e yields the postcondition of the method that was forked to obtain token e. We assume that the receiver and arguments of the fork are stored in the token, but omit this aspect in the encoding.

condition $\mathcal{B}[\textbf{term}] < \mathit{term}$ expresses that the callee promises to terminate. In this case, exhaling its precondition will transfer a terminates-obligation from the caller to the callee, that is, decrease the value of $\mathcal{B}[\textbf{term}]$ compared to the value before the exhale (stored in local variable *term*). Finally, since the assertion after the exhale quantifies over all obligations, including terminates-obligations, it enforces that the callee promises to terminate if the caller does (otherwise the caller would still hold its terminates-obligation after the exhale). Since terminates-obligations must be satisfied by each individual method and cannot be delegated, we restore the terminates-obligations after the exhale. The final inhale does not have to record measures since this is necessary only in the prestate of a method execution or loop iteration; therefore, it uses the dummy map P_d, which is never read from.

Allowing the caller to retain obligations when calling a terminating method is crucial for modularity; otherwise, the callee's precondition would have to mention different obligations for different call sites. Nevertheless, these obligations are accounted for in variable *residue* and, thus, affect wait level constraints. In particular, it is not possible for a caller to hold an obligation to unblock its callee (which might create a deadlock) because the obligation in the caller affects the wait level of the callee (via *residue*) and, thus, prevents the callee from executing the blocking operation (see for instance the first assertion in the encoding of join statements in Fig. 11).

The encoding of a fork statement is similar to a call. In particular, the measures of transferred obligation must decrease to ensure that they cannot be transferred from thread to thread indefinitely. However, since the forked method will be executed in a new thread, there are no restrictions on the obligations that remain in the forker. After the exhale, we pick a fresh token for the new thread. The fact that this token is different from existing

$$
\begin{array}{lcl}
[\![v := \texttt{new lock}]\!] & = & \textbf{havoc } v; \textbf{assume } \mathcal{L}[v] = \bot; \\
& & \textbf{havoc } w; \textbf{assume } \mathit{levelBelow}(\mathcal{B}, w); \\
& & \mathcal{L}[v] := w; \\
& & \mathcal{B}[v] := 0; \mathcal{F}[v] := 0; \\
[\![\texttt{acquire } e]\!] & = & \textbf{assert } \mathit{levelBelow}(\mathcal{B}, [\![e]\!]); \\
& & \mathit{Inhale}_{obl}([\![e]\!], 1, \top, P_d) \\
[\![\texttt{release } e]\!] & = & \mathit{Exhale}_{obl}([\![e]\!], 1, \bot, P_\top)
\end{array}
$$

Figure 12 Encoding of lock operations.

token is encoded by assuming that its level in the wait order is $\bot$, whereas all tokens for existing threads are implicitly assumed to have larger levels. The new thread can be joined if it promises to terminate, that is, if the forker's terminates-obligations get decreased by exhaling the forked method's precondition. Like for calls, the terminates-obligations get restored afterwards. Finally, we choose a wait level for the new thread that is above the current wait level, which will allow the current thread to join it later.

Since join is a blocking operation, it asserts that the token of the thread to be joined is above the current wait level (to avoid deadlock) and that the current thread has the appropriate join-permission (to avoid waiting on a non-terminating thread). We then inhale the joined method's postcondition and remove the join-permission to prevent a thread from being joined more than once, which could forge credits in the postcondition.

5.3 Lock Operations

The encoding of lock operations is presented in Fig. 12. To focus on the essentials, we do not associate locks with an invariant. An extension is straightforward, but requires that the invariant does not contain obligations (credits are permitted) [15]. Otherwise, a thread could get rid of its obligations by storing them in a lock, which might never get acquired again.

Creating a new lock picks a fresh **lock** value. The fact that this value is different from existing locks is encoded by assuming that its level in the wait order is $\bot$, whereas all other locks are assumed to have larger levels. The new lock is then inserted into the wait order above the current wait level, which allows the current thread to acquire it (specifying different levels for the new lock is possible [15], but omitted here for simplicity). Initially, the current thread does not hold any obligations for the new lock.

Acquiring a lock checks that the lock is above the current wait level to prevent deadlock. It then inhales a fresh releases-obligation for the lock to ensure that the acquired lock will eventually be released. Inhaling this obligation implicitly raises the current thread's wait level.

Releasing a lock exhales the corresponding releases-obligation. This exhale operation does not have to check that the obligation measure has been decreased. We achieve that by passing a non-$\top$ measure for the obligation (here, $\bot$) and $P_\top$ for the prestate map, such that the decrease-check succeeds trivially (since $\bot \sqsubset \top$).

$$
\begin{array}{lcl}
[\![v := \texttt{new}\ C]\!] & = & \textbf{havoc}\ v; \textbf{assume}\ \mathcal{L}[v] = \bot; \\
& & \textbf{havoc}\ w; \textbf{assume}\ \mathit{levelBelow}(\mathcal{B}, w); \\
& & \mathcal{L}[v] := w; \\
& & \mathcal{B}[v] := 0; \mathcal{F}[o] := 0 \\
\\
[\![\texttt{receive}\ v := e]\!] & = & \textbf{assert}\ \mathit{levelBelow}(\mathcal{B}, [\![e]\!]); \\
& & \textbf{assert}\ \mathcal{B}[[\![e]\!]] < 0; \\
& & \mathit{Exhale}_{obl}([\![e]\!], -1, \bot, \_) \\
& & \mathit{Inhale}(\mathit{inv}([\![e]\!], v), P_d) \\
\\
[\![\texttt{send}\ e_1(e_2)]\!] & = & \mathit{Exhale}_{obl}([\![e_1]\!], 1, \bot, P_\top) \\
& & \mathit{Exhale}(\mathit{inv}([\![e_1]\!], [\![e_2]\!]), P_\top)
\end{array}
$$

Figure 13 Encoding of channel operations. C is a channel type, and *inv* denotes its invariant, which may refer to the channel itself, for instance, to denote sends-credits for the channel.

5.4 Message Passing

The encoding of channel operations is presented in Fig. 13. Channel creation is analogous to lock creation (see Fig. 12). Since receive is a blocking operation, we first assert that the channel is above the current wait level. Moreover, to ensure that some thread has an obligation to send on the channel (or has sent already), we require that the current thread has a sends-credit, which is subsequently consumed by exhaling it. Finally, we inhale the channel invariant *inv*, without recording any obligations measures. Since we assume sending to be a non-blocking operation, we simply exhale a sends-obligation (which got satisfied) and exhale the channel invariant.

Our well-formedness conditions (Sec. 3.2) ensure that channel invariants do not contain obligations. Therefore, it is neither possible to get rid of obligations by sending them in a message that is never received, nor to send obligations in circles indefinitely. It is also not possible to transfer obligations indirectly from one thread to another by sending a credit in the opposite direction. Such an indirect transfer would have to cancel the obligation in the receiver of the message with the credit contained in the message, which is prevented by our definition of inhale (see Sec. 4.2).

5.5 Loops

The encoding of loops (Fig. 14) includes both the representation of the loop within the enclosing code and the verification of the loop body. The two aspects are encoded by a non-deterministic choice (`if(*)`). The former resembles the encoding of a method call, whereas the latter is similar to a method body.

In both cases, we proceed by exhaling the loop invariant. This exhale does not check obligation measures since the measures of the loop encoded here are independent of the measures used by the enclosing method or loop (if any). Hence, we pass the all-top map $P_\top$ to the exhale operation. The check after exhaling the loop invariant ensures that the code before the loop does not retain any obligations unless the loop promises to terminate. This assertion is identical to the one for calls (Fig. 11). In particular, it enforces that the loop must promise to terminate if the enclosing loop or method has a terminates-obligation. We

$[\![\texttt{while}(e)\ \texttt{invariant}\ A\ \{\ S\ \}]\!] =$
  var *term* := $\mathcal{B}$[**term**];
  Exhale($A, P_\top$)
  assert $\forall o \in \textbf{obl} \cdot \mathcal{B}[o] \leq 0 \vee \mathcal{B}[\textbf{term}] < term$;
  havoc *loop targets*;
  if $(*)$ {
    Inhale(A, P_d)
    assume $\neg[\![e]\!]$;
  } **else** {
    havoc $\mathcal{B}, \mathcal{F}$, *residue*;
    assume $\forall o \in \textbf{obl} \cdot \mathcal{B}[o] = 0$;
    var $P_{loop} := P_\top$;
    Inhale(A, P_{loop})
    foreach $o \in \textbf{obl}$ { $\mathcal{F}[o] := 0$; }
    assume $[\![e]\!]$;
    $[\![S]\!]$
    Exhale(A, P_{loop})
    assert $\forall o \in \textbf{obl} \cdot \mathcal{B}[o] \leq 0$;
    assume `false`;
  }

Figure 14 Encoding of while statements. The first branch of the non-deterministic choice encodes the loop within the enclosing code and resembles a method call. The second branch verifies the loop body and resembles the encoding of a method.

then havoc the loop targets, that is, all local variables that get assigned to in the loop body. Any information about these variables that should be retained must be included in the loop invariant.

To represent the loop within the enclosing code, we simply inhale the loop invariant (without recording obligation measures), assume that the loop condition is false, and proceed to the statements after the loop.

To verify the loop body, we consider an arbitrary loop iteration. We first havoc the obligation maps and *residue* to remove any information from before the loop. The following steps are analogous to the encoding of methods (Fig. 10): Before inhaling the loop invariant, the loop iteration holds neither obligations nor credits. Then we inhale the loop invariant and record obligation measures in a map P_{loop} for the decrease-check at the end of the loop body. Finally, we make all fresh obligations non-fresh (to prevent them from being transferred indefinitely from iteration to iteration), and execute the loop body. After the loop body, we exhale the loop invariant, checking that obligation measures decreased during the loop body, and perform the same leak check as for methods. Finally, we stop verification by assuming false, in order to prevent verification from proceeding with the code after the loop (which is done in the other branch of the non-deterministic choice).

6 Soundness

Our technique guarantees that in any execution of a verified program, no thread blocks indefinitely. This guarantee holds under the assumptions that (1) all thread transitions are

strongly fair and (2) the number of threads in each execution state is finite. A strongly-fair transition executes infinitely often if it is enabled infinitely often. Hence, we make the assumption that the thread scheduler ensures strong fairness and that we have fair locks and fair message reception. The number of threads in each state must be finite to prevent infinite chains of threads where each thread is blocked by its successor and, thus, never gets unblocked. This requirement is met by any execution platform with finite memory; to implement it, a fork operation aborts the entire program execution when a certain (unknown) number of threads is reached. Note that the number of threads is finite, but unbounded. That is, verification guarantees finite blocking for program executions with an arbitrary finite number of threads in each state. In this section, we provide the main arguments why our technique is sound.

The following properties hold in each execution state of a verified program:

1. *A thread t holds a lock l iff t has a releases-obligation for l.* This property is preserved by all lock operations (Fig. 12). The other operations preserve it because they neither add nor remove releases-obligations. In particular, our well-formedness conditions (Sec. 3.2) ensure that releases-obligations cannot be transferred to another thread during fork, join, or message passing.
2. *For each channel c, the total number of credits in the system (that is, held by a thread or stored in a message) is at most the total number of obligations plus the number of messages stored in c's buffer.* This inequality is preserved by all channel operations (Fig. 13). For all other operations, each exhale has a corresponding inhale, keeping the total number of obligations and credits in the system constant. The only exception is exhaling the postcondition of a forked method if the thread does not get joined. However, our well-formedness conditions ensure that postconditions of forked methods do not contain obligations. Moreover, our leak checks ensure that the termination of method executions and loop iterations maintains the number of obligations in the system and does not increase the number of credits, thus, preserving the inequality.
3. *If a thread t has a join-permission for a thread t' then t' has a terminates-obligation or has terminated already.* Fork and join (Fig. 11) preserve the property. In particular, fork provides a join-permission only if the new thread promises to terminate, and join removes this permission. Moreover, a thread keeps its terminates-obligation until the forked method terminates.
4. *If a thread t is blocked, the number of obligations to unblock it held by all other threads is positive.* This property follows from the encoding of the three blocking statements and Properties 1–3.
5. *There is no cycle among threads such that each thread on the cycle waits for the next one to unblock it; that is, there is no deadlock.* Each blocking statement checks that the wait level of the current thread is strictly less than the wait level of the thread that must unblock it, that is, the thread that (a) holds the lock to be acquired (since held locks contribute to the wait level by Property 1), (b) has a sends-obligation for the channel on which to receive, or (c) needs to terminate (since the thread's current wait level is no smaller than its initial wait level, which is the level of its token).

The following properties hold for each execution of a verified program:

6. *A fresh obligation gets satisfied or becomes non-fresh within finitely many execution steps.* A single thread t can hold on to a fresh obligations only for a finite number of steps because every fresh obligation becomes non-fresh at the beginning of each method or loop body, that is, before the thread can transfer the obligations to another thread.

7. *A non-fresh obligation gets satisfied within finitely many execution steps.* A non-fresh obligation cannot stay in one thread forever since its measure must decrease for each recursive call or loop iteration. It can be transferred to other threads only via fork, which also checks that the measure decreases. The well-formedness conditions ensure that transfers through join or message passing are not possible.

These properties imply soundness as follows. Whenever there is a blocked thread t_0 then there is a sequence $t_0, t_1, \ldots$ such that t_{i+1} has an obligation to unblock t_i. By the assumption that the number of threads is finite, this sequence is finite. By Properties 4 and 5, its last thread t_n is not blocked, that is, is enabled. By the assumption of fair scheduling, t_n will eventually make progress and, by Properties 6 and 7, its obligation will eventually be satisfied, unblocking thread t_{n-1}. Thread t_{n-1} might re-block immediately if another thread acquires the lock or receives the message t_{n-1} is waiting for. However, since we assume fair locks and message reception, enabling t_{n-1} infinitely often ensures that it will make progress eventually. Therefore, the argument applies inductively.

7 Related Work

Chalice. The work most closely related to ours is Leino et al.'s approach to verifying deadlock freedom in Chalice [15]. However, their verification technique uses a partial correctness semantics and, thus, provides no guarantees for the common case that a program contains non-terminating threads. It also does not support termination proofs. In contrast, the key contribution of our work is a technique to prove finite blocking even in the presence of non-terminating threads, and this technique subsumes termination checking. Leino et al. handle blocking receive statements via credits and obligations (called debt). We generalize this idea to arbitrary blocking operations, which gives us a uniform treatment of locks, channels, and thread join, and provides a systematic way to encode further blocking operations. This uniform treatment also allows us to replace several ad-hoc solutions in Chalice such as holds-predicates and lockchange-clauses [14]. We adopted the general approach of preventing deadlock via a wait order that includes locks, channels, and threads from Chalice. However, the encoding of wait level constraints presented by Leino et al. is unsound because it does not interpret `waitlevel` consistently during exhale and inhale. Our encoding fixes this problem via the 2-phase exhale and a consistent interpretation during inhale.

Liveness. Finite blocking and termination are liveness properties that can be proved using linear-time temporal logic [17]. For instance, Manna et al. [18] verify liveness properties of concurrent programs running an arbitrary number of (identical) threads. In contrast to this work, we present a methodology based on obligations that provides a strategy how to structure specifications and proofs. In particular, our technique supports modular verification, where each method is verified without knowledge of their callers or concurrently executing threads. Like our work, Manna et al. use strong fairness as one of their fairness notions.

Gotsman et al. [8] present a verification technique to show that a non-blocking algorithm is wait-free, lock-free, or obstruction-free. These liveness properties are checked by proving termination of an arbitrary number of operations running in parallel. The authors use a rely-guarantee logic to reason about the interference between these parallel executions, which is non-modular. Our work focuses on blocking operations. In this context, we can use specifications based on obligations and credits to make verification modular.

Model checkers are able to verify general temporal logic properties in LTL or CTL, including liveness properties. Many model checkers bound the number of threads (such as

SPIN [9]) or the number of context switches (such as CHESS [19]), whereas our technique verifies programs for any finite number of threads and any number of context switches. Software model checking can also be applied to infinite state programs by utilizing different (automatic) abstraction techniques [2]. In contrast to these approaches, our technique is procedure-modular, which makes it applicable to libraries and improves scalability, at the price of having to write specifications.

Termination. Our technique is closely related to existing work on termination checking. However, it goes beyond termination checking in two major ways. First, it allows one to prove finite blocking in concurrent programs, which includes termination checking as a special case. In particular, finite blocking requires a solution that distinguishes safe implementations where a thread unblocks another thread and then obtains yet another obligation to unblock (for instance, by releasing and re-acquiring a lock) from unsafe situations where a thread continues to block another thread. Such situations do not occur during termination checking. Second, our technique handles different kinds of obligations and supports the dual notion of credit. In particular, credits may be transferred between threads, which requires extra checks to prevent unsound cancellation. Again, this problem does not occur in termination checking.

Le et al. [12] propose a verification logic for termination and non-termination. Similar to our work, their logic uses a resource that reflects termination and that is manipulated similarly to permissions in permission logics. Le et al. associate their termination resources with upper and lower bounds on their lifetimes, which allows them to prove termination as well as definite non-termination.

We adopted Dafny's approach to obtain measures by defining a well-founded order on all values of a program execution [13]. Dafny lifts this order to define a lexicographic order on sequences of values and includes the import relation among modules as a part of this order. These extensions are compatible with our use of measures.

There exist powerful automated termination checkers for both sequential and concurrent programs [2, 3, 4, 6]. The focus of most work in this area is on inferring termination measures. By contrast, we assume the measure to be provided by the programmer and use it to prove finite blocking. Combining our work with inference techniques is an interesting direction for future work, especially in the presence of credits.

Deadlock freedom. There are numerous verification techniques and type systems to check deadlock freedom of programs that either synchronize via locks [7, 11, 24] or communicate via messages [5, 10]. Our work adopts Chalice's solution to checking deadlock freedom, and we refer to Leino et al. [15] for a detailed comparison to related work. The contribution of our work is to recast the Chalice solution in a uniform framework that supports a variety of blocking operations and to fix the soundness issue in Chalice that we mentioned above.

8 Conclusion

This paper introduces a novel verification technique to prove finite blocking in concurrent programs. At its core is a general framework for obligations, which express that a thread must perform a certain operation eventually. We present uniform proof rules for the manipulation of obligations and use them to encode three common blocking operations, which are representative for the various characteristics of obligations. By associating obligations with measures, our technique guarantees finite blocking even for programs containing non-

terminating threads under the assumption that scheduling, locks, and message receipt are strongly fair. Our technique subsumes termination checking and integrates verification of deadlock freedom.

As future work, we plan to use additional kinds of obligations to remove the main limitations or our technique. For instance, one could allow sending obligations over channels by introducing another form of obligation to ensure that every sent message will eventually be received such that the contained obligations do not get lost. Obligations to establish conditions on shared state could be used to prove that the busy-wait loop of a thread terminates. However, obligations are not limited to finite blocking. We plan to use the framework introduced here to prove other liveness properties, for instance, that every asynchronous task will be awaited eventually or that certain objects will be de-allocated eventually. Another direction for future work is to add support for abstract predicates [20] in order to denote statically-unknown sets of obligations in specifications, and to support information hiding. Finally, it would also be interesting to combine our work with approaches to infer termination measures.

Acknowledgments. We would like to Alex Summers for various discussions. We are grateful to the anonymous reviewers for their valuable comments. Pontus Boström was partially funded by a scholarship from Svenska kulturfonden. Peter Müller's work was funded in part by the Hasler Foundation.

References

1 M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

2 B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *POPL*, pages 265–276. ACM, 2007.

3 B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI*, pages 320–330. ACM, 2007.

4 B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Commun. ACM*, 54(5), 2011.

5 M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In D. Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

6 P. Ganty and S. Genaim. Proving termination starting from the end. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 397–412. Springer, 2013.

7 C. S. Gordon, M. D. Ernst, and D. Grossman. Static lock capabilities for deadlock freedom. In *TLDI*, pages 67–78. ACM, 2012.

8 A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, pages 16–28. ACM, 2009.

9 G. J. Holzmann. *SPIN Model Checker, The: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

10 N. Kobayashi. A new type system for deadlock-free processes. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

11 D.-K. Le, W.-N. Chin, and Y.-M. Teo. An expressive framework for verifying deadlock freedom. In D. Van Hung and M. Ogawa, editors, *ATVA*, volume 8172 of *LNCS*, pages 287–302. Springer, 2013.

12 T. C. Le, C. Gherghina, A. Hobor, and W.-N. Chin. A resource-based logic for termination and non-termination proofs. In S. Merz and J. Pang, editors, *ICFEM*, volume 8829 of *LNCS*, pages 267–283. Springer, 2014.

13 K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

14 K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *ESOP*, volume 5502 of *LNCS*, pages 378–393. Springer, 2009.

15 K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In A. D. Gordon, editor, *ESOP*, volume 6012 of *LNCS*, pages 407–426. Springer, 2010.

16 Z. Manna and A. Pnueli. A hierarchy of temporal properties. In C. Dwork, editor, *PODC*, pages 377–410. ACM, 1990.

17 Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.

18 Z. Manna and A. Pnueli. Verification of parameterized programs. *Specification and Validation Methods*, pages 167–230, 1995.

19 M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455. ACM, 2007.

20 M. J. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM, 2005.

21 M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. In G. Barthe, editor, *ESOP*, volume 6602 of *LNCS*, pages 439–458. Springer, 2011.

22 J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society Press, 2002.

23 J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In S. Drossopoulou, editor, *ECOOP*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.

24 K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In G. Ramalingam, editor, *APLAS*, volume 5356 of *LNCS*, pages 155–170. Springer, 2008.

A Credits and Deadlock Freedom

As explained in Sec. 2.1, blocking operations where the very first execution blocks (such as receiving on a channel) are handled by credits. Creating a credit simultaneously creates a corresponding obligation. Therefore, by enforcing that a thread executing a receive statement holds a sends-credit, we ensure that some other thread has a sends-obligation and, thus, the receive will not block indefinitely.

The producer-consumer example in Fig. 15 demonstrates this idea. The consumer method `Cons` requires a sends-credit for the channel `c`, which allows it to receive one message on this channel. Because of `Cons`'s precondition, this initial credit is provided by the `Main` method when it forks the consumer, which leaves the corresponding sends-obligation in `Main`. This obligation is transferred to the producer when the `Prod` method is forked. Note that `Main` could not terminate without forking the producer first because it would still hold an obligation and, thus, not pass the leak check at the end of the method. Note further that neither the producer nor the consumer promise to terminate and, thus, cannot be joined since the join operation might block indefinitely.

Once the producer and consumer have been forked, they communicate via the channel `c`. The declaration of `c`'s type `C` specifies that messages sent over the channel are boolean values.

```
channel C(b: bool) where b ⇒ sends(this, -1, ⊤);

method Main() {
  c := new C;
  fork t1 := Cons(c) below c;
  fork t2 := Prod(c);
}

method Prod(c: C)
  requires sends(ch, 1, 1);
{
  while(*)
    invariant sends(c, 1, 1);
  { send c(true); }
  send c(false);
}

method Cons(c: C)
  requires sends(c, -1, ⊤)
  requires waitlevel ≪ c;
{
  more := true;
  while(more)
    invariant more ⇒ sends(c, -1, 1);
    invariant waitlevel ≪ c;
  { receive more := c; }
}
```

Figure 15 A producer-consumer example. The producer and consumer communicate over an asynchronous channel `c`. The main method transfers a sends-credit to the consumer, which allows it to receive the first message, and the corresponding sends-obligation to the producer, forcing it to send a message. With every message except the final one, the producer sends another sends-credit to the consumer, which allows the consumer to receive the next message. The measure $\top$ is explained in Sec. 3.2.

Its channel invariant expresses that whenever the value true is sent over the channel, the message includes one sends-credit for the channel. Therefore, with every send operation inside the while loop of method `Prod`, the producer sends a credit to the consumer. Consequently, the producer has one sends-obligation throughout the loop because it satisfies one obligation by sending a message and obtains a new one by sending away a credit. This property is expressed by its loop invariant. Since the sends-obligation gets satisfied in each loop iteration, its measure is constant 1. However, similar examples require other measures; for instance, if the producer sent messages to several channels in a round-robin fashion, the sends-obligation for each of the channels would be the number of channels. Once the loop has terminated, the producer sends a final message not containing a credit. This send operation satisfies the remaining sends-obligation, allowing method `Prod` to pass its leak check and terminate. The consumer obtains another credit with every message it receives, which allows it to receive the next message. The final message (with value false) contains no credit, forcing the consumer to terminate its receive-loop.

To prevent deadlock, the receive operation in the consumer requires that the consumer's wait level is strictly below the level of the channel c. This constraint is required in the precondition and maintained throughout the loop. In order to satisfy the precondition, method `Main` forks the consumer with an initial wait level that is below c's level (indicated by the `below`-clause). We omit such constraints from our encoding, but their treatment is straightforward [15].