Deadlock-Free Monitors

Jafar Hamin and Bart Jacobs

imec-DistriNet, Dept. C.S., KU Leuven
Celestijnenlaan 200A, 3001 Heverlee, Belgium
{jafar.hamin, bart.jacobs}@cs.kuleuven.be

Abstract. Monitors constitute one of the common techniques to synchronize threads in multithreaded programs, where calling a wait command on a condition variable suspends the caller thread and notifying a condition variable causes the threads waiting for that condition variable to resume their execution. One potential problem with these programs is that a waiting thread might be suspended forever leading to deadlock, a state where each thread of the program is waiting for a condition variable or a lock. In this paper, a modular verification approach for deadlock-freedom of such programs is presented, ensuring that in any state of the execution of the program if there are some threads suspended then there exists at least one thread running. The main idea behind this approach is to make sure that for any condition variable v for which a thread is waiting there exists a thread obliged to fulfil an obligation for v that only waits for a waitable object whose wait level, an arbitrary number associated with each waitable object, is less than the wait level of v. The relaxed precedence relation introduced in this paper, aiming to avoid cycles, can also benefit some other verification approaches, verifying deadlock-freedom of other synchronization constructs such as channels and semaphores, enabling them to accept a wider range of deadlock-free programs. We encoded the proposed proof rules in the VeriFast program verifier and by defining some appropriate invariants for the locks associated with some condition variables succeeded in verifying some popular use cases of monitors including unbounded/bounded buffer, sleeping barber, barrier, and readers-writers locks. A soundness proof for the presented approach is provided; some of the trickiest lemmas in this proof have been machine-checked with Coq.

1 Introduction

One of the popular mechanisms for synchronizing threads in multithreaded programs is using monitors, a synchronization construct allowing threads to have mutual exclusion and also the ability to wait for a certain condition to become true. These constructs, consisting of a mutex/lock and some condition variables, provide some basic functions for their clients, namely $\mathsf{wait}(v, l)$, causing the calling thread to wait for the condition variable v and release lock l while doing so, and $\mathsf{notify}(v)$/$\mathsf{notifyAll}(v)$, causing one/all thread(s) waiting for v to resume their execution. Each condition variable is associated with a lock; a thread must acquire the associated lock for waiting or notifying on a condition variable, and when a thread is notified it must reacquire the associated lock.

However, one potential problem with these synchronizers is deadlock, where all threads of the program are waiting for a condition variable or a lock. To clarify the problem consider the program in Figure 1, where a channel consists of a queue q, a lock l and a condition variable v, protecting a thread from dequeuing q when it is empty. In this program the receiver thread first acquires lock l and while there is no item in q it releases l, suspends itself and waits for a notification on v. If this thread is notified while q is not empty it dequeues an item and finally releases l. The sender thread also acquires the same lock, enqueues an item into q, notifies one of the threads waiting for v, if any, and lastly releases l. After creating a channel ch, the main thread of the program first forks a thread to receive a message from ch and then sends a message on ch. Although this program is deadlock-free, it is easy to construct some variations of it that lead to deadlock: if the main thread itself, before sending any messages, tries to receive a message from ch, or if the number of receives is greater than the number of sends, or if the receiver thread waits for v even if q is not empty.

```
routine main()              routine send(channel ch, int d)   routine receive(channel ch)
{q := newqueue;             {acquire(ch.l);                    {acquire(ch.l);
 l := newlock;               enqueue(ch.q, d);                  while(sizeof(ch.q) = 0)
 v := newcond;               notify(ch.v);                         wait(ch.v, ch.l);
 ch := channel(q, l, v);     release(ch.l)}                     d := dequeue(ch.q);
 fork (receive(ch));                                            release(ch.l);
 send(ch, 12)}                                                  d}
```

Fig. 1. A message passing program synchronized using a monitor

Several approaches to verify termination, deadlock-freedom, liveness, and finite blocking of threads of programs have been presented. Some of these approaches only work with non-blocking algorithms [1,2,3], where the suspension of one thread cannot lead to the suspension of other threads. These approaches are not applicable for condition variables because suspension of a sender thread in Figure 1, for example, might cause a receiver thread to be blocked forever. Some other approaches are also presented to verify termination of programs using some blocking constructs such as channels [4,5,6] and semaphores [7]. These approaches are not general enough to cover condition variables because unlike the channels and semaphores a notification of a condition variable is lost when there is no thread waiting for that condition variable. There are also some studies [8,9,10] to verify correctness of programs that support condition variables. However, these approaches either only cover a very specific application of condition variables, such as a buffer program with only one producer and one consumer, or are not modular and suffer from a long verification time when the size of the state space, such as the number of threads, is increased.

In this paper we present a modular approach to verify deadlock-freedom of programs in the presence of condition variables. More specifically, this approach makes sure that for any condition variable v for which a thread is waiting there exists a thread obliged to fulfil an obligation for v that only waits for a waitable object whose wait level, an arbitrary number associated with each waitable object, is less than the wait level of v. The presented approach is modular, meaning that different modules (functions) of a program can be verified individually. This approach is based on Leino *et al.* [4] approach for verification of deadlock-freedom in the presence of channels and locks, which in turn was based on Kobayashi's [6] type system for verifying deadlock-freedom of π-calculus processes, and extends the separation logic-based encoding [11] by covering condition variables. We implemented the proposed proof rules in the VeriFast verifier [12,13,14] and succeeded in verifying some common applications of condition variables such as bounded/unbounded buffer, sleeping barber [15], barrier, and readers-writers locks (see the full version of this paper [16] reporting the verification time of these programs).

This paper is structured as follows. Section 2 provides some background information on the existing approaches upon which we build our verification algorithm. Section 3 introduces a preliminary approach for verifying deadlock-freedom of some common applications of condition variables. In Section 4 the precedence relation, aiming to avoid cycles, is relaxed, making it possible to verify some trickier applications of condition variables. A soundness proof of the presented approach is lastly given in Section 5.

2 Background Information on the Underlying Approaches

In this section we provide some background information on the existing approaches that verify absence of data races and deadlock in the presence of locks and channels that we build on.

2.1 Verifying Absence of Data Races

Locks/mutexes are mostly used to avoid data races, an undesired situation where a heap location is being written and accessed concurrently by two different threads. One common approach to verify absence of these undesired conditions is ownership: ownership of heap locations is assigned to threads and it is verified that a thread accesses only the heap locations that it owns. Transferring ownership of heap locations between threads is supported through locks by allowing locks, too, to own heap locations. While a lock is not held by a thread, it owns the heap locations described by its *invariant*. More specifically, when a lock is created the resources specified by its invariant are transferred from the creating thread to the lock, when that lock is acquired these resources are transferred from the lock to the acquiring thread, and when that lock is released these resources, that must be again in possession of the thread, are again transferred from the thread to the lock [17]. Figure 2 illustrates how a program increasing a counter, which consists of an integer variable x and a lock l protecting this variable, can be verified, where two threads try to write on the variable x. We use separation logic [18] to reason about the ownership of permissions. As indicated below each command, creating the integer variable x initialized by zero provides a read/write access permission to x, denoted by $x \mapsto 0$. This ownership, that is going to be protected by lock l, is transferred to the lock because it is asserted by the lock invariant inv, which is associated with the lock, as denoted by function I, at the point where the lock is initialized. The resulting lock permission, that can be duplicated, is used in the routine inc, where x is increased under protection of lock l. Acquiring this lock in this routine provides a full access permission to x and transforms the lock permission to a locked permission, implying that the related lock has been acquired. Releasing that lock again consumes this access permission and transforms the locked permission to a lock one.

```
$x$:=newint(0);
{$x \mapsto 0$}
$l$ := newlock;
{ulock($l$) $*$ $x \mapsto 0$}
$ct$ := counter($x$:=$x$, $l$:=$l$);
{ulock($ct.l$) $*$ $ct.x \mapsto 0$}
{ulock($ct.l$) $*$ inv($ct$)}
{lock($ct.l$) $\wedge$ I($l$)=inv($ct$)}
{lock($ct.l$) $*$ lock($ct.l$)}
fork (inc($ct$));
{lock($ct.l$)}
inc($ct$)
```

```
routine inc(counter $ct$){
 {lock($ct.l$) $\wedge$ I($l$)=inv($ct$)}
 acquire($ct.l$);
 {locked($ct.l$) $*$ $\exists z.\ ct.x \mapsto z$}
 $ct.x$:=$ct.x$+1;
 {locked($ct.l$) $*$ $\exists z.\ ct.x \mapsto z$}
 release($ct.l$)
 {lock($ct.l$)}}
```

Fig. 2. Verification of data-race-freedom of a program, where $\mathsf{inv}{=}\lambda ct.\ \exists z.\ ct.x \mapsto z$

2.2 Verifying Absence of Deadlock

One potential problem with programs using locks and other synchronization mechanisms is deadlock, an undesired situation where all threads of the program are waiting for some waitable objects. For example, a program can deadlock if a thread acquires a lock and forgets to release it, because any other thread waiting for that lock never succeeds in acquiring that lock. As another example, if in a message passing program the number of threads trying to receive a message from a channel is greater than the number of messages sent on that channel there will be some threads waiting for that channel forever. One approach to verify deadlock-freedom of channels and locks is presented by Leino *et al.* [4] that guarantees deadlock-freedom of programs by ensuring that 1) for any *obligee* thread waiting for a waitable object, such as a channel or lock, there is an *obligation* for that object that must be fulfilled by an *obligor* thread, where a thread can fulfil an obligation for a channel/lock if it sends a message on that channel/releases that lock, and 2) each thread waits for an object only if the *wait level* of that object, an arbitrary number assigned to each waitable object, is lower than the wait levels of all obligations of that thread. The second rule is established by making sure that when a thread with some obligations O executes a command $\mathsf{acquire}(o)/\mathsf{receive}(o)$ the precondition $o \prec O$ holds, i.e. the wait level of o is lower than the wait levels of obligations in O. To meet the first rule where the waitable object is a lock, as the example in the left side of Figure 3 illustrates, after acquiring a lock, that lock is loaded onto the bag [1] (multiset) of obligations of the thread, denoted by $\mathsf{obs}(O)$. This ensures that if a thread tries to acquire a lock that has already been acquired then there is one thread obliged to fulfil an obligation for that lock.

```
{obs(O) ∗ lock(l) ∧ l≺O}
acquire(l);
{obs(O⊎{l}) ∗ locked(l) ∗ I(l)}
...
{obs(O⊎{l}) ∗ locked(l) ∗ I(l)}
release(l)
{obs(O) ∗ lock(l)}
```

```
{obs(O)}
{obs(O⊎{ch}) ∗ credit(ch)}
fork (
 {obs({}) ∗ credit(ch) ∧ ch≺{}}
 receive(ch)
 {obs({})}
);
{obs(O⊎{ch})}
send(ch, 12) {obs(O)}
```

Fig. 3. Verification of deadlock-freedom of locks (left side) and channels (right side)

To establish the first rule where the waitable object is a channel any thread trying to receive a message from a channel ch must spend one *credit* for ch. This credit is normally obtained from the thread that has forked the receiver thread, where this credit is originally created by loading ch onto the bag of obligations of the forking thread. The forking thread can discharge the loaded obligation by either sending a message on the corresponding channel or delegating it to a child thread that can discharge it. The example on the right side of Figure 3 shows the verification of deadlock-freedom a program in which the main routine, after forking a obligee thread trying to receive a message from channel ch, sends a message on this channel. Before forking the receiver thread, a credit and an obligation for the channel ch are created in the main thread. The former is given to the forked thread, where this credit is spent by the $\mathsf{receive}(ch)$ command, and the latter is fulfilled by the main thread when it executes the command $\mathsf{send}(ch, 12)$.

More formally, the mentioned verification approach satisfies the first rule by ensuring that for each channel ch in the program the number of obligations for ch is equal to/greater than the number of threads waiting for ch. This assurance is obtained by preserving the invariant $Wt(ch)+Ct(ch) \leqslant Ot(ch)+\mathsf{sizeof}(ch)$, while the programming language itself ensures that $\mathsf{sizeof}(ch)>0 \Rightarrow Wt(ch)=0$,

[1] We treat bags of waitable objects as functions from waitable objects to natural numbers.

where sizeof is a function mapping each channel to the size of its queue, $Wt(ch)$ is the total number of threads currently waiting for channel ch, $Ot(ch)$ is the total number of obligations for channel ch held by all threads, and $Ct(ch)$ is the total number of credits for channel ch currently in the system.

2.3 Proof Rules

The separation logic-based proof rules, introduced by Jacobs *et al.* [11], avoiding data races and deadlock in the presence of locks and channels are shown in Figure 4, where R and I are functions mapping a waitable object/lock to its wait level/invariant, respectively, and g_initl, and g_load are some *ghost commands* used to initialize an uninitialized lock permission and load a channel onto the bag of obligations and credits of a thread, respectively. When a lock is created, as shown in NewLock, an uninitialized lock permission ulock(l) is provided for that thread. Additionally, an arbitrary integer number z can be decided as the wait level of that lock that is stored in R. Note that variable z in this rule is universally quantified over the rule, and different applications of the NewLock rule can use different values for this variable. The uninitialized lock permission, as shown in InitLock, can be converted to a normal lock permission lock(l) provided that the resources described by the invariant of that lock, stored in I, that must be in possession of the thread, are transferred from the thread to the lock. By the rule Acquire, having a lock permission, a thread can acquire that lock if the wait levels of obligations of that thread are all greater than the wait level of that lock. After acquiring the lock, the resources represented by the invariant of that lock are provided for the acquiring thread and the permission lock is converted to a locked permission. When a thread releases a lock, as shown in the rule Release, the resources indicated by the invariant of that lock, that must be in possession of the releasing thread, are transferred from the thread to the lock and the permission locked is again converted to a lock permission. By the rule Receive a thread with obligations O can try to receive a message from a channel ch only if the wait level of ch is lower than the wait levels of all obligations in O. This thread must also spend one credit for ch, ensuring that there is another thread obliged to fulfil an obligation for ch. As shown in the rule Send, an obligation for this channel can be discharged by sending a message on that channel. Alternatively, by the rule Fork, a thread can discharge an obligation for a channel if it delegates that obligation to a child thread, provided that the child thread discharges the delegated obligation. In this setting the verification of a program starts with an empty bag of obligations and must also end with such bag implying that there is no remaining obligation to fulfil.

However, this verification approach is not straightforwardly applicable to condition variables. A command notify cannot be treated like a command send because a notification on a condition variable is lost when there is no thread waiting for that variable. Accordingly, it does not make sense to discharge an obligation for a condition variable whenever it is notified. Similarly, a command wait cannot be treated like a command receive. A command wait is normally

$$\textsc{NewLock}\quad \{\mathsf{true}\}\ \mathsf{newlock}\ \{\lambda l.\ \mathsf{ulock}(l) \land \mathsf{R}(l){=}z\}$$

$$\textsc{InitLock}\quad \{\mathsf{ulock}(l) * i\}\ \mathsf{g\_initl}(l)\ \{\lambda\_.\ \mathsf{lock}(l) \land \mathsf{I}(l){=}i\}$$

$$\textsc{Acquire}\ \{\mathsf{lock}(l) * \mathsf{obs}(O) \land l{\prec}O\}\ \mathsf{acquire}(l)\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{\!\!\{l\}\!\!\}) * \mathsf{locked}(l) * \mathsf{I}(l)\}$$

$$\textsc{Release}\ \{\mathsf{obs}(O) * \mathsf{locked}(l) * \mathsf{I}(l)\}\ \mathsf{release}(l)\ \{\lambda\_.\ \mathsf{obs}(O{-}\{\!\!\{l\}\!\!\}) * \mathsf{lock}(l)\}$$

$$\textsc{NewChannel}\quad \{\mathsf{true}\}\ \mathsf{newchannel}\ \{\lambda ch.\ \mathsf{R}(ch){=}z\}$$

$$\textsc{Send}\quad \{\mathsf{obs}(O)\}\ \mathsf{send}(ch, v)\ \{\lambda\_.\ \mathsf{obs}(O{-}\{\!\!\{ch\}\!\!\})\}$$

$$\textsc{Receive}\quad \{\mathsf{obs}(O) * \mathsf{credit}(ch) \land ch{\prec}O\}\ \mathsf{receive}(ch)\ \{\lambda\_.\ \mathsf{obs}(O)\}$$

$$\textsc{Fork}\ \frac{\{a * \mathsf{obs}(O)\}\ c\ \{\lambda\_.\ \mathsf{obs}(\{\!\!\{\}\!\!\})\}}{\{a * \mathsf{obs}(O{\uplus}O')\}\ \mathsf{fork}(c)\ \{\lambda\_.\ \mathsf{obs}(O')\}} \qquad \textsc{DupLock}\quad \mathsf{lock}(l) \Leftrightarrow \mathsf{lock}(l) * \mathsf{lock}(l)$$

$$\textsc{LoadOb}\ \{\mathsf{obs}(O)\}\ \mathsf{g\_load}(ch)\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{\!\!\{ch\}\!\!\}) * \mathsf{credit}(ch)\}$$

Fig. 4. Proof rules ensuring deadlock-freedom of channels and locks, where $o{\prec}O \Leftrightarrow \forall o'{\in}O.\ \mathsf{R}(o){<}\mathsf{R}(o')$

executed in a while loop, checking the *waiting condition* of the related condition variable. Accordingly, it is impossible to build a loop invariant for such a loop if we force the wait command to spend a credit for the related condition variable.

3 Deadlock-Free Monitors

3.1 High-Level Idea

In this section we introduce an approach to verify deadlock-freedom of programs in the presence of condition variables. This approach ensures that the verified program never deadlocks, i.e. there is always a running thread, that is not blocked, until the program terminates. The main idea behind this approach is to make sure that for any condition variable v for which a thread is waiting there exists a thread obliged to fulfil an obligation for v that only waits for a waitable object whose wait level is less than the wait level of v. As a consequence, if the program has some threads suspended, waiting for some obligations, there is always a thread obliged to fulfil the obligation o_{min} that is not suspended, where o_{min} has a minimal wait level among all waitable objects for which a thread is waiting. Accordingly, the proposed proof rules make sure that 1) when a command $\mathsf{wait}(v, l)$ is executed $Ot(v){>}0$, where Ot maps each condition variable v to the total number of obligations for v held by all threads (note that having a thread with permission $\mathsf{obs}(O)$ implies $O(v){\leqslant}Ot(v)$), 2) a thread discharges an obligation for a condition variable only if after this discharge the invariant

$\mathsf{one\_ob}(v, Wt, Ot)$ defined as $Wt(v){>}0 \Rightarrow Ot(v){>}0$ still holds, where $Wt(v)$ denotes the number of threads waiting for condition variable v, and 3) a thread with obligations O executes a command $\mathsf{wait}(v, l)$ only if $v{\prec}O$.

3.2 Tracking Numbers of Waiting Threads and Obligations

For all condition variables associated with a lock l the value of functions Wt and Ot can only be changed by a thread that has locked l; $Wt(v)$ is changed only when one of the commands $\mathsf{wait}(v, l)/\mathsf{notify}(v)/\mathsf{notifyAll}(v)$ is executed, requiring holding lock l, and we allow $Ot(v)$ to be changed only when a permission locked for l is available. Accordingly, when a thread acquires a lock these two bags are stored in the related locked permission and are used to establish the rules number 1 and 2, when a thread executes a wait command or discharges one of its obligations. Note that the domain of these functions is the set of the condition variables associated with the related lock. The thread executing the critical section can change these two bags under some circumstances. If that thread loads/discharges a condition variable onto/from the list of its obligations this condition variable must also be loaded/discharged onto/from the bag Ot stored in the related locked permission. Note that unlike the approach presented by Leino *et al.* [4], an obligation for a condition variable can arbitrarily be loaded or discharged by a thread, provided that the rule number 2 is respected. At the start of the execution of a $\mathsf{wait}(v, l)$ command, $Wt(v)$ is incremented and after execution of commands $\mathsf{notify}(v)/\mathsf{notifyAll}(v)$ one/all instance(s) of v is/are removed from the bag Wt stored in the related locked permission, since these commands change the number of threads waiting for v.

A program can be successfully verified according to the mentioned rules, formally indicated in Figure 5, if each lock associated with any condition variable v has an appropriate invariant such that it implies the desired invariant $\mathsf{one\_ob}(v, Wt, Ot)$. Accordingly, the proof rules allow locks to have invariants parametrized over the bags Wt and Ot. When a thread acquires a lock the result of applying the invariant of that lock to these two bags, stored in the related locked permission, is provided for the thread and when that lock is released it is expected that the result of applying the lock invariant to those bags, stored in the related locked permission, again holds. However, before execution of a command $\mathsf{wait}(v, l)$, when lock l with bags Wt and Ot stored in its locked permission is going to be released, it is expected that the invariant of l holds with bags $Wt \uplus \{\!\{v\}\!\}$ and Ot because the running thread is going to wait for v and this condition variable is going to be added to Wt. As this thread resumes its execution, when it has some bags Wt' and Ot' stored in the related locked permission, the result of applying the invariant of l to these bags is provided for that thread. Note that the total number of threads waiting for v, $Wt(v)$, is already decreased when a command $\mathsf{notify}(v)$ or $\mathsf{notifyAll}(v)$ is executed, causing the waiting thread(s) to wake up and try to acquire the lock associated with v.

3.3 Resource Transfer on Notification

In general, as we will see when looking at examples, it is sometimes necessary to transfer resources from a notifying thread to the threads being notified [2]. To this end, these resources, specified by a function M, are associated with each condition variable v when v is created, such that the commands $\mathsf{notify}(v)$/$\mathsf{notifyAll}(v)$ consume one/$Wt(v)$ instance(s) of these resources, respectively, and the command $\mathsf{wait}(v, l)$ produces one instance of such resources (see the rules Wait,Notify, and NotifyAll in Figure 5).

3.4 Proof Rules

Figure 5 shows the proposed proof rules used to verify deadlock-freedom of condition variables, where L and M are functions mapping each condition variable to its associated lock and to the resources that are moved from the notifying thread to the notified one when that condition variable is notified, respectively. Creating a lock, as shown in the rule NewLock, produces a permission ulock storing the bags Wt and Ot, where these bags are initially empty. The bag Ot in this permission, similar to a locked one, can be changed provided that the obligations of the running thread are also updated by one of the ghost commands $\mathsf{g\_chrg}(v)$ or $\mathsf{g\_disch}(v)$ (see rules ChargeOb and DisOb). The lock related to this permission can be initialized by transferring the resources described by the invariant of this lock, that is now parametrized over the bags Wt and Ot, applied to the bags stored in this permission from the thread to the lock (see rule InitLock). When this lock is acquired, as shown in the rule Acquire, the resources indicated by its invariant are provided for the thread, and when it is released, as shown in the rule Release, the resources described by its invariant that must hold with appropriate bags, are again transferred from the thread to the lock. The rules Wait and DisOb ensure that for any condition variable v when the number of waiting threads is increased, by executing a command $\mathsf{wait}(v, l)$, or the number of the obligations is decreased, by (logically) executing a command $\mathsf{g\_disch}(v)$, the desired invariant $\mathsf{one\_ob}$ still holds. Additionally, the rules Acquire and Wait make sure that a thread only waits for a waitable object whose wait level is lower that the wait levels of obligations of that thread. Note that in the rule Wait in the precondition of the command $\mathsf{wait}(v, l)$ it is not necessary that the wait level of v is lower that the wait level of l, since lock l is going to be released by this command. However, in this precondition the wait level of l must be lower that the wait levels of the obligations of the thread because when this thread is notified it tries to reacquire l, at which point $l \prec O$ must hold. The commands $\mathsf{notify}(v)$/$\mathsf{notifyAll}(v)$, as shown in the rules Notify and NotifyAll, remove one/all instance(s) of v, if any, from the bag Wt stored in the related locked permission. Additionally, $\mathsf{notify}(v)$ consumes the moving resources, indicated by $\mathsf{M}(v)$, that appear in the postcondition of the notified

[2] This transfer is only sound in the absence of spurious wake-ups, where a thread is awoken from its waiting state even though no thread has signaled the related condition variable.

thread. Note that $\mathsf{notifyAll}(v)$ consumes $Wt(v)$ instances of these resources, since they are transferred to $Wt(v)$ threads waiting for v.

$$\textsc{NewLock}\ \{\mathsf{true}\}\ \mathsf{newlock}\ \{\lambda l.\ \mathsf{ulock}(l, \{\}, \{\}) \wedge \mathsf{R}(l){=}z\}$$

$$\textsc{NewCv}\ \{\mathsf{true}\}\ \mathsf{newcond}\ \{\lambda v.\ \mathsf{R}(v){=}z \wedge \mathsf{L}(v){=}l \wedge \mathsf{M}(v){=}m\}$$

$$\textsc{Acquire}\ \begin{array}{c}\{\mathsf{lock}(l) * \mathsf{obs}(O) \wedge l{\prec}O\}\ \mathsf{acquire}(l)\\ \{\lambda\_.\ \exists Wt, Ot.\ \mathsf{locked}(l,\ Wt, Ot) * \mathsf{I}(l)(Wt, Ot) * \mathsf{obs}(O{\uplus}\{l\})\}\end{array}$$

$$\begin{array}{l}\textsc{Release}\\ \{\mathsf{locked}(l,\ Wt, Ot) * \mathsf{I}(l)(Wt, Ot) * \mathsf{obs}(O{\uplus}\{l\})\}\ \mathsf{release}(l)\ \{\lambda\_.\ \mathsf{lock}(l) * \mathsf{obs}(O)\}\end{array}$$

$$\textsc{Wait}\ \begin{array}{c}\{\mathsf{locked}(l,\ Wt, Ot) * \mathsf{I}(l)(Wt{\uplus}\{v\}, Ot) * \mathsf{obs}(O{\uplus}\{l\})\\ \wedge\ l{=}\mathsf{L}(v) \wedge v{\prec}O \wedge l{\prec}O \wedge \mathsf{safe\_obs}(v,\ Wt{\uplus}\{v\}, Ot)\}\ \mathsf{wait}(v, l)\\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{l\}) * \exists Wt', Ot'.\ \mathsf{locked}(l,\ Wt', Ot') * \mathsf{I}(l)(Wt', Ot') * \mathsf{M}(v)\}\end{array}$$

$$\textsc{Notify}\ \begin{array}{c}\{\mathsf{locked}(\mathsf{L}(v),\ Wt, Ot) * (Wt(v) = 0 \vee \mathsf{M}(v))\}\ \mathsf{notify}(v)\\ \{\lambda\_.\ \mathsf{locked}(\mathsf{L}(v),\ Wt{-}\{v\}, Ot)\}\end{array}$$

$$\begin{array}{l}\textsc{NotifyAll}\\ \{\mathsf{locked}(\mathsf{L}(v),\ Wt, Ot) * (\mathop{*}\limits_{i:=0}^{Wt(v)} \mathsf{M}(v))\}\ \mathsf{notifyAll}(v)\ \{\lambda\_.\ \mathsf{locked}(\mathsf{L}(v),\ Wt[v{:=}0], Ot)\}\end{array}$$

$$\begin{array}{l}\textsc{InitLock}\\ \{\mathsf{ulock}(l,\ Wt, Ot) * inv(Wt, Ot) * \mathsf{obs}(O)\}\ \mathsf{g\_initl}(l)\ \{\lambda\_.\ \mathsf{lock}(l) * \mathsf{obs}(O) \wedge \mathsf{I}(l){=}inv\}\end{array}$$

$$\textsc{ChargeOb}\ \begin{array}{c}\{\mathsf{obs}(O) * \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot)\}\ \mathsf{g\_chrg}(v)\\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{v\}) * \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot{\uplus}\{v\})\}\end{array}$$

$$\textsc{DisOb}\ \begin{array}{c}\{\mathsf{obs}(O) * \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot) \wedge \mathsf{safe\_obs}(v,\ Wt(v), Ot{-}\{v\})\}\\ \mathsf{g\_disch}(v)\ \{\lambda\_.\ \mathsf{obs}(O{-}\{v\}) * \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot{-}\{v\})\}\end{array}$$

Fig. 5. Proof rules to verify deadlock-freedom of condition variables, where $Wt(v)$ and $Ot(v)$ denote the total number of threads waiting for v and the total number of obligations for v, respectively, and $\mathsf{safe\_obs}(v,\ Wt, Ot) \Leftrightarrow \mathsf{one\_ob}(v,\ Wt, Ot)$ and $\mathsf{one\_ob}(v,\ Wt, Ot) \Leftrightarrow (Wt(v){>}0 \Rightarrow Ot(v){>}0)$

3.5 Verifying Channels

Ghost Counters. We will now use our proof system to prove deadlock-freedom of the program in Figure 1. To do so, however, we will introduce a *ghost resource* that plays the role of *credits*, in such a way that we can prove the invariant $Wt(ch){+}Ct(ch) \leqslant Ot(ch){+}\mathsf{sizeof}(ch)$. In particular, we want this property to follow from the lock invariant. This means we need to be able to talk, in the lock invariant, about the total number of credits in the system. To achieve this,

$$\textsc{NewCounter}\ \{\mathsf{true}\}\ \mathsf{g\_newctr}\ \{\lambda c.\ \mathsf{ctr}(c, 0)\}$$

$$\textsc{IncCounter}\ \{\mathsf{ctr}(c, n)\}\ \mathsf{g\_inc}(c)\ \{\lambda\_.\ \mathsf{ctr}(c, n{+}1) * \mathsf{tic}(c)\}$$

$$\textsc{DecCounter}\ \{\mathsf{ctr}(c, n) * \mathsf{tic}(c)\}\ \mathsf{g\_dec}(c)\ \{\lambda\_.\ \mathsf{ctr}(c, n{-}1) \land 0{<}n\}$$

Fig. 6. Ghost counters

we introduce a notion of *ghost counters* and corresponding *ghost counter tickets*, both of which are a particular kind of ghost resources. Specifically, we introduce three ghost commands: g_newctr, g_inc, and g_dec. g_newctr allocates a new ghost counter whose *value* is zero and returns a *ghost counter identifier* c for it. g_inc(c) increments the value of the ghost counter with identifier c and produces a *ticket* for the counter. g_dec(c), finally, consumes a ticket for ghost counter c and decrements the ghost counter's value. Since these are the only operations that manipulate ghost counters or ghost counter tickets, it follows that the value of a ghost counter c is always equal to the number of tickets for c in the system. Proof rules for these ghost commands are shown in Figure 6 [3].

The Channels Proof. Figure 7 illustrates how the program in Figure 1 can be verified using our proof system. The invariant of lock $ch.l$ in this program, denoted by inv(ch), is parametrized over bags Wt, Ot and implies the desired invariant one_ob($ch.v$, Wt, Ot). The permission ctr($ch.c, Ctv$) in this invariant indicates that the total number of credits (tickets) for $ch.v$ is Ctv, where $ch.c$ is a *ghost field* added to the channel data structure, aiming to store a ghost counter identifier for the ghost counter of $ch.v$. Generally, a lock invariant can imply the invariant one_ob(v, Wt, Ot) if it asserts $Wt(v){+}Ct(v){\leqslant}Ot(v){+}S(v)$ and $Wt(v){\leqslant}Ot(v)$, where $Ct(v)$ is the total number of credits for v and $S(v)$ is an integer value such that the command wait(v, l) is executed only if $S(v){\leqslant}0$. After initializing l in the main routine, there exists a credit for $ch.v$ (denoted by tic($ch.c$)) that is consumed by the thread executing the receive routine, and also an obligation for $ch.v$ that is fulfilled by this thread after executing the send routine. The credit tic($ch.c$) in the precondition of the routine receive ensures that before execution of the command wait($ch.v, ch.l$), $Ot(ch.v){>}0$. This inequality follows from the invariant of lock l, which holds for $Wt{\uplus}\{\!\{ch.v\}\!\}$ and Ot when Ctv is decreased by g_dec($ch.c$). This credit (or the one specified by M($ch.v$) that is moved from a notifier thread when the receiver thread wakes up) must be consumed after execution of the command dequeue($ch.q$) and before releasing $ch.l$ to make sure that the invariant still holds after decreasing the number of items in $ch.q$. The obligation for $ch.v$ in the precondition of the routine send

[3] Some logics for program verification, such as Iris [19], include general support for defining ghost resources such as our ghost counters. In particular, our ghost counters can be obtained in Iris as an instance of the *authoritative monoid* [19, p. 5].

```
inv(channel $ch$) ::= $\lambda Wt.\ \lambda Ot.\ \exists Ctv.$ ctr($ch.c, Ctv$) $*\ \exists s.$ queue($ch.q, s$) $\wedge$
 L($ch.v$)=$ch.l$ $\wedge$ M($ch.v$)=tic($ch.c$) $\wedge$
 $Wt(ch.v) + Ctv \leqslant Ot(ch.v) + s \wedge$
 $Wt(ch.v) \leqslant Ot(ch.v)$

routine main(){{obs({})}
 $q$:=newqueue; $l$:=newlock; $v$:=newcond; $c$:=g_newctr; g_inc($c$);
 {obs({}) $*$ ulock($l$, {}, {}) $*$ queue($q$, 0) $*$ ctr($c$, 1) $*$ tic($c$)
 $\wedge$ L($v$)=$l$ $\wedge$ M($v$)=tic($c$) $\wedge$ R($l$)=0 $\wedge$ R($v$)=1}
 $ch$:=channel($q, l, v$); $ch.c$:=$c$;
 {obs({}) $*$ ulock($l$, {}, {}) $*$ inv($ch$)({}, {$v$}) $*$ tic($c$)} g_chrg($v$);
 {obs({$v$}) $*$ ulock($l$, {}, {$v$}) $*$ inv($ch$)({}, {$v$}) $*$ tic($c$)} g_initl($l$);
 {obs({$v$}) $*$ lock($l$) $*$ tic($c$) $\wedge$ I($l$)=inv($ch$)}
 fork (receive($ch$));
 {obs({$v$}) $*$ lock($l$)}
 send($ch$, 12) {obs({})}}

routine receive(channel $ch$){
 {obs($O$) $*$ tic($ch.c$) $*$ lock($ch.l$) $\wedge$ $ch.l \prec O$ $\wedge$ $ch.v \prec O$ $\wedge$ I($ch.l$)=inv($ch$)}
 acquire($ch.l$);
 {obs($O \uplus${$ch.l$}) $*$ tic($ch.c$) $*\ \exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot$)}
 while(sizeof($ch.q$) = 0){ g_dec($ch.c$);
   {obs($O \uplus${$ch.l$}) $*\ \exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt \uplus${$ch.v$}, $Ot$)}}
   wait($ch.v, ch.l$)
   {obs($O \uplus${$ch.l$}) $*$ M($ch.v$) $*\ \exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot$)}};
 dequeue($ch.q$); g_dec($ch.c$);
 {obs($O \uplus${$ch.l$}) $*\ \exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot$)}
 release($ch.l$) {obs($O$) $*$ lock($ch.l$)}}

routine send(channel $ch$, int $d$){
 {obs($O \uplus${$ch.v$}) $*$ lock($ch.l$) $\wedge$ $ch.l \prec O \uplus${$ch.v$} $\wedge$ I($ch.l$)=inv($ch$)}
 acquire($ch.l$);
 {obs($O \uplus${$ch.v, ch.l$}) $*\ \exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot$)}
 enqueue($ch.q, d$);
 if ($Wt(ch.v)>0$) g_inc($ch.c$);
 notify($ch.v$);
 {obs($O \uplus${$ch.v, ch.l$}) $*\ \exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot-${$ch.v$})}
 g_disch($ch.v$);
 {obs($O \uplus${$ch.l$}) $*\ \exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot$)}
 release($ch.l$) {obs($O$) $*$ lock($ch.l$)}}
```

Fig. 7. Verification of the program in Figure 1

is discharged by this routine, which is safe, since after the execution of the commands $\mathsf{enqueue}$ and notify the invariant $\mathsf{one\_ob}(ch.v, Wt, Ot{-}\{\!\!\{ch.v\}\!\!\})$, which follows from the lock invariant, holds.

3.6 Other Examples

Using the proof system of this section we prove two other deadlock-free programs, namely *sleeping barber* [16], and *barrier*. In the barrier program shown in Figure 8, a barrier b consists of an integer variable r indicating the number of the remaining threads that must call the routine $\mathsf{wait\_for\_rest}$, a lock l protecting r against data races, and a condition variable v. Each thread executing the routine $\mathsf{wait\_for\_rest}$ first decreases the variable r, and if the resulting value is still positive waits for v, otherwise it notifies all threads waiting for v. In this program the barrier is initialized to 3, implying that no thread must start task_2 unless all the three threads in this program finish task_1. This program is deadlock-free because the routine $\mathsf{wait\_for\_rest}$ is executed by three different threads. Figure 8 illustrates how this program can be verified by the presented proof rules. Note that before executing $\mathsf{g\_disch}$ in the else branch, $\mathsf{safe\_obs}$ holds because at this point we have $0{<}b.r$, which implies $1{<}b.r$ before the execution of $b.r{:=}b.r{-}1$, and by the invariant we have $1{<}Ot(b.v)$, implying $0{<}(Ot - \{\!\!\{b.v\}\!\!\})(b.v)$. The interesting point about the verification of this program is that since all the threads waiting for condition variable v in this program are notified by the command $\mathsf{notifyAll}$, the invariant of the related lock, implying $\mathsf{one\_ob}(b.v, Wt, Ot)$, is significantly different from the ones defined in the channel and sleeping barber examples. Generally, for a condition variable v on which only $\mathsf{notifyAll}$ is executed (and not notify) a lock invariant can imply the invariant $\mathsf{one\_ob}(v, Wt, Ot)$ if it asserts $Wt(v){=}0 \vee S(v){\leqslant}Ct(v)$ and $Ct(v){<}Ot(v){+}S(v)$, where $Ct(v)$ is the total number of credits for v and $S(v)$ is an integer value such that the command $\mathsf{wait}(v, l)$ is executed only if $S(v){\leqslant}0$. For this particular example $S(b.v){=}1{-}b.r$ and $Ct(b.v){=}0$, since this program can be verified without incorporating the notion of credits.

4 Relaxing the Precedence Relation

The precedence relation, in this paper denoted by $\prec$, introduced in [4] makes sure that all threads wait for the waitable objects in strict ascending order (with respect to the wait level associated with each waitable object), or here in this paper in descending order, ensuring that in any state of the execution there is no cycle in the corresponding wait-for graph. However, this relation is too restrictive and prevents verifying some programs that are actually deadlock-free, such as the one shown in the left side of Figure 9. In this program a value is increased by two threads communicating through a channel. Each thread receives a value from the channel, increases that value, and then sends it back on the channel. Since an initial value is sent on the related channel this program is deadlock-free. The first attempt to verify this program is illustrated in the middle part of Figure 9,

```
routine main(){
 $r$:=newint(3);
 $l$:=newlock;
 $v$:=newcond;
 $b$:=barrier($r, l, v$);
 fork (task$_1$(); wait_for_rest($b$); task$_2$());
 fork (task$_1$(); wait_for_rest($b$); task$_2$());
 task$_1$(); wait_for_rest($b$); task$_2$()}
```

```
routine wait_for_rest(barrier $b$){
 acquire($b.l$);
 $b.r$:=$b.r-1$;
 if($b.r$=0)
   notifyAll();
 else
   while($b.r>0$)
      wait($b.v, b.l$);
 release($b.l$)}
```

$$\mathsf{inv}(\mathsf{barrier}\ b) ::= \lambda Wt.\ \lambda Ot.\ \exists r \geqslant 0.\ b.r \mapsto r \wedge \mathsf{L}(b.v)=b.l \wedge \mathsf{M}(b.v)=\mathsf{true} \wedge \\ (Wt(b.v)=0 \vee 0<r) \ \wedge\ (r \leqslant Ot(b.v))$$

```
routine main(){$\{\mathsf{obs}(\{\})\}$
 $r$:=newint(3); $l$:=newlock; $v$:=newcond;
 $\{\mathsf{obs}(\{\}) * r\mapsto 3 * \mathsf{ulock}(l, \{\}, \{\}) \wedge \mathsf{L}(v)=l \wedge \mathsf{M}(v)=\mathsf{true} \wedge \mathsf{R}(l)=0 \wedge \mathsf{R}(v)=1\}$
 $b$:=barrier($r, l, v$);
 $\{\mathsf{obs}(\{\}) * \mathsf{inv}(b)(\{\}, \{3\cdot v\}) * \mathsf{ulock}(l, \{\}, \{\})\}$
 g_chrg($v$); g_chrg($v$); g_chrg($v$); g_initl($l$);
 $\{\mathsf{obs}(\{3\cdot v\}) * \mathsf{lock}(l) \wedge \mathsf{I}(l)=\mathsf{inv}(b)\}$
 fork (wait_for_rest($b$));
 $\{\mathsf{obs}(\{2\cdot v\}) * \mathsf{lock}(l)\}$
 fork (wait_for_rest($b$));
 $\{\mathsf{obs}(\{v\}) * \mathsf{lock}(l)\}$
 wait_for_rest($b$) $\{\mathsf{obs}(\{\})\}$}
```

```
routine wait_for_rest(barrier $b$){
 $\{\mathsf{obs}(O \uplus \{b.v\}) * \mathsf{lock}(b.l) \wedge b.l \prec O \uplus \{b.v\} \wedge b.v \prec O \wedge \mathsf{I}(b.l)=\mathsf{inv}(b)\}$
 acquire($b.l$);
 $\{\mathsf{obs}(O \uplus \{b.v, b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l, Wt, Ot) * \mathsf{inv}(b)(Wt, Ot)\}$
 $b.r$:=$b.r-1$;
 if($b.r$=0){
   notifyAll($b.v$);
   $\{\mathsf{obs}(O \uplus \{b.v, b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l, Wt[b.v:=0], Ot)$
   $*\mathsf{inv}(b)(Wt[b.v:=0], Ot-\{b.v\})\}$ g_disch($b.v$)
   $\{\mathsf{obs}(O \uplus \{b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l, Wt, Ot) * \mathsf{inv}(b)(Wt, Ot)\}$}
 else{
   $\{\mathsf{obs}(O \uplus \{b.v, b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l, Wt, Ot)$
   $*\mathsf{inv}(b)(Wt, Ot-\{b.v\})\}$ g_disch($b.v$);
   $\{\mathsf{obs}(O \uplus \{b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l, Wt, Ot) * \mathsf{inv}(b)(Wt, Ot)\}$
   while($b.r>0$)
     $\{\mathsf{obs}(O \uplus \{b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l, Wt, Ot) * \mathsf{inv}(b)(Wt \uplus \{b.v\}, Ot)\}$
     wait($b.v, b.l$)
     $\{\mathsf{obs}(O \uplus \{b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l, Wt, Ot) * \mathsf{inv}(b)(Wt, Ot)\}$};
 release($b.l$) $\{\mathsf{obs}(O) * \mathsf{lock}(b.l)\}$}
```

Fig. 8. Verification of a barrier synchronized using a monitor

```
routine main(){
   ch:=channel;
   send(ch, 12);
   fork (inc(ch));
   fork (inc(ch))}

routine inc(channel ch){
 d:=receive(ch);
 send(ch, d+1)}
```

```
routine main(){
 {obs({})}
 ch:=newchannel;
 send(ch, 12);
 fork (inc(ch));
 fork (inc(ch)) {obs({})}}

routine inc(channel ch){
 {obs({})}
 {obs({ch}) ∗ credit(ch)
 ∧ ch⊀{ch}}
 d:=receive(ch);
 {obs({ch})}
 send(ch, d+1) {obs({})}}
```

```
routine main(){
 {obs({})}
 ch:=newchannel;
 {obs({ch}) ∧ P(ch)=true}
 send(ch, 12);
 {obs({})}
 fork (inc(ch));
 fork (inc(ch)) {obs({})}}

routine inc(channel ch){
 {obs({}) ∧ ch≼{ch}}
 ⟨obs({ch}) ∗ credit(ch)
 ∧ ch≼{ch}⟩
 d:=receive(ch);
 {obs({ch})}
 send(ch, d+1) {obs({})}}
```

Fig. 9. A deadlock-free program verified by exploiting the relaxed precedence relation

where the required credit to verify the receive command in the routine inc is going to be provided by the send command, executed immediately after this command, and not by the precondition of this routine. In other words, the idea is to load a credit and an obligation for ch in the routine inc itself, and then spend the loaded credit to verify the receive(ch) command and fulfil the loaded obligation by the send(ch) command. However, this idea fails because the receive command in the routine inc cannot be verified since one of its preconditions, $ch \prec \{\!\{ch\}\!\}$, never holds. Kobayashi [20,6] has addressed this problem in his type system by using the notion of *usages* and assigning levels to each *obligation/capability*, instead of waitable objects. However, in the next section we provide a novel idea to address this problem by just relaxing the precedence relation used in the presented proof rules.

4.1 A Relaxed Precedence Relation

To tackle the problem mentioned in the previous section we relax the precedence relation, enforced by $\prec$, by replacing $\prec$ by $\preccurlyeq$ satisfying the following property: $o \preccurlyeq O$ holds if either $o \prec O$ or 1) $o \prec O - \{\!\{o\}\!\}$, and 2) o satisfies the property that in any execution state, if a thread waits for o then there exists a thread that can discharge an obligation for o and is not waiting for any object whose wait level is equal to/greater than the wait level of o. This property still guarantees that in any state of the execution if the program has some threads suspended, waiting for some obligations, there is always a thread obliged to fulfil the obligation o_{min} that is not blocked, where o_{min} has a minimal wait level among all waitable objects for which a thread is waiting.

The condition number 2 is met if it is an invariant that for a condition variable o for which a thread is waiting the total number of obligations is greater than the total number of waiting threads. Since each thread waiting for o has at most one instance of o in the bag of its obligations, according to the *pigeonhole principle*, if the number obligations for o is higher than the number of threads waiting for o then there exists a thread that holds an obligation for o that is not waiting for o, implying the rule number 2 because this thread only waits for objects whose wait levels are lower than the wait level of o. Accordingly, we first introduce a new function P in the proof rules mapping each waitable object to a boolean value, and then make sure that for any object o for which a thread is waiting if $\mathsf{P}(o)=\mathsf{true}$ then $Wt(o)<Ot(o)$. With the help of this function we define the relaxed precedence relation as shown in Definition 1.

Definition 1 (Relaxed precedence relation). *The relaxed precedence relation indexed over functions* R *and* P *holds for a waitable object* v *and a bag of obligations* O*, denoted by* $v \preccurlyeq O$*, if and only if:*

$$v \prec O \ \vee\ (v \prec O - \{\!\!\{v\}\!\!\} \wedge \mathsf{P}(v)=\mathsf{true}) \text{ , where } v \prec O \Leftrightarrow \forall o \in O.\ \mathsf{R}(v)<\mathsf{R}(o)$$

Using this relaxed precedence relation the approach presented by Leino *et al.* [4] can also support more complex programs, such as the one in the left side of Figure 9. This approach can exploit this relation by 1) replacing the original precedence relation $\prec$ by the relaxed one $\preccurlyeq$, and 2) replacing the rule associated with creating a channel by the one shown below. According to this proof rule for each channel ch the function P, in the definition of the relaxed precedence relation, is initialized when ch is created such that if $\mathsf{P}(ch)$ is decided to be true then one obligation for ch is loaded onto the bag of obligations of the creating thread. The approach is still sound because for any channel ch for which P is true the invariant $Wt(ch)+Ct(ch)<Ot(ch)+\mathsf{sizeof}(ch)$ holds. Combined with the fact that in this language, where channels are primitive constructs, $Wt(ch)>0 \Rightarrow \mathsf{sizeof}(ch)=0$, we have $Wt(ch)>0 \Rightarrow Wt(ch)<Ot(ch)$. Now consider a deadlocked state, where each thread is waiting for a waitable object. Among all of these waitable objects take the one having a minimal wait level, namely o_m. If o_m is a lock or a channel, where $\mathsf{P}(o_m)=\mathsf{false}$, then at least one thread has an obligation for o_m and is waiting for an object o whose wait level is lower that the wait level of o_m, which contradicts minimality of the wait level of o_m. Otherwise, since $Wt(o_m)>0$ we have $Wt(o_m)<Ot(o_m)$. Additionally, we know that each thread waiting for o_m has at most one obligation for o_m. Accordingly, there must be a thread holding an obligation for o_m that is not waiting for o_m. Consequently, this thread must be waiting for an object o whose wait level is lower than the wait level of o_m, which contradicts minimality of the wait level of o_m.

$$\{\mathsf{obs}(O)\}\ \mathsf{newchannel}\ \{\lambda ch.\ \mathsf{obs}(O') \wedge \mathsf{R}(ch)=z \wedge \mathsf{P}(ch)=b$$
$$\wedge((b=\mathsf{false} \wedge O'=O) \vee (b=\mathsf{true} \wedge O'=O \uplus \{\!\!\{ch\}\!\!\}))\}$$

To exploit the relaxed definition in the approach presented in this paper we only need to make sure that for any condition variable v for which a thread is waiting if $\mathsf{P}(v)$ is true then $Ot(v)$ is greater than $Wt(v)$. To achieve this goal we include this invariant in the definition of the invariant $\mathsf{safe\_obs}$, shown in Definition 2, an invariant that must hold when a command wait or a ghost command $\mathsf{g\_disch}$ is executed.

Definition 2 (Safe Obligations). *The relation* safe_obs(v, Wt, Ot)*, indexed over function* P*, holds if and only if:*

$\mathsf{one\_ob}(v, Wt, Ot) \wedge (\mathsf{P}(v)=\mathsf{true} \Rightarrow \mathsf{spare\_ob}(v, Wt, Ot))$*, where*

$\mathsf{one\_ob}(v, Wt, Ot) \Leftrightarrow (Wt(v)>0 \Rightarrow Ot(v)>0)$

$\mathsf{spare\_ob}(v, Wt, Ot) \Leftrightarrow (Wt(v)>0 \Rightarrow Wt(v)<Ot(v))$

```
routine main(){
 $aw$:=newint(0);
 $ww$:=newint(0);
 $ar$:=newint(0);
 $l$:=newlock;
 $v_w$:=newcond;
 $v_r$:=newcond;
 $b$ := rdwr($aw, ww$
  $, ar, l, v_w, v_r$);
 fork(
  while (true)
   fork(reader($b$))
 );
 while (true)
  fork(writer($b$))
}
```

```
routine reader(rdwr $b$){
 acquire($b.l$);
 while($b.aw$+$b.ww$>0)
   wait($b.v_r, b.l$);
 $b.ar$:=$b.ar$+1;
 release($b.l$);
 // Perform reading ...
 acquire($b.l$);
 if($b.ar$<1)
   abort;
 $b.ar$:=$b.ar$−1;
 notify($b.v_w$);
 release($b.l$)}
```

```
routine writer(rdwr $b$){
 acquire($b.l$);
 while($b.aw$+$b.ar$>0){
   $b.ww$:=$b.ww$+1;
   wait($b.v_w, b.l$);
   if($b.ww$<1)
      abort();
   $b.ww$:=$b.ww$−1
 };
 $b.aw$:=$b.aw$+1;
 release($b.l$);
 // Perform writing ...
 acquire($b.l$);
 if($b.aw$≠1)
   abort;
 $b.aw$:=$b.aw$−1;
 notify($b.v_w$);
 if($b.ww$=0)
   notifyAll($b.v_r$);
 release($b.l$)}
```

Fig. 10. A readers-writers program with variables aw, holding the number of threads writing, ww, holding the number of thread waiting to write, and ar, holding the number of threads reading, that is synchronized using a monitor consisting of condition variables v_w, preventing writers from writing while other threads are reading or writing, and v_r, preventing readers from reading while there is another thread writing or waiting to write.

Readers-Writes Locks. As another application of this relaxed definition consider a readers-writers program, shown in Figure 10 [4], where the condition variable v_w prevents writers from writing on a shared memory when that memory is being accessed by other threads. After reading the shared memory, a reader thread notifies this condition variable if there is no other thread reading that memory. This condition variable is also notified by a writer thread when it finishes its writing. Consequently, a writer thread first might wait for v_w and then fulfil an obligation for this condition variable. This program is verified if the writer thread itself produces a credit and an obligation for v_w and then uses the

[4] The abort commands in this program can be eliminated using the ghost counters from Figure 6. However, we leave them in for simplicity.

```
$\mathsf{inv}(\mathsf{rdwr}\ b) ::= \lambda Wt.\ \lambda Ot.\ \exists Ctw.\ \mathsf{ctr}(b.c_w, Ctw) *$
 $\exists aw \geqslant 0, ww \geqslant 0, ar \geqslant 0.\ b.aw \mapsto aw * b.ww \mapsto ww * b.ar \mapsto ar \wedge$
 $\mathsf{L}(b.v_w)=\mathsf{L}(b.v_r)=b.l \wedge \mathsf{M}(b.v_w)=\mathsf{tic}(b.c_w) \wedge \mathsf{M}(b.v_r)=\mathsf{true} \wedge \mathsf{P}(v_w)=\mathsf{true} \wedge \mathsf{P}(v_r)=\mathsf{false} \wedge$
 $(Wt(b.v_r) = 0 \vee 0 < aw + ww) \wedge$
 $aw + ww \leqslant Ot(b.v_r) \wedge$
 $Wt(b.v_w) + Ctw + aw + ar \leqslant Ot(b.v_w) \wedge$
 $(Wt(b.v_w) = 0 \vee Wt(b.v_w) < Ot(b.v_w))$
```

```
routine main(){
 $aw$:=newint(0); $ww$:=newint(0);
 $ar$:=newint(0); $l$:=newlock;
 $v_w$:=newcond; $v_r$:=newcond;
 $b$ := rdwr($aw, ww, ar, l, v_w, v_r$);
 $b.c_w$:=g_newctr;
 {obs({}) * inv($b$)({}, {}) * ulock($l$, {}, {}) *
 $\mathsf{L}(v_w)=\mathsf{L}(v_r)=l \wedge \mathsf{M}(v_w)=\mathsf{tic}(b.c_w) \wedge$
 $\mathsf{M}(v_r)=\mathsf{true} \wedge \mathsf{R}(l)=0 \wedge \mathsf{R}(v_w)=1 \wedge$
 $\mathsf{R}(v_r)=2 \wedge \mathsf{L}(v_w)=l \wedge \mathsf{L}(v_r)=l$
 $\wedge\ \mathsf{P}(v_w)=\mathsf{true} \wedge \mathsf{P}(v_r)=\mathsf{false}$} g_initl($l$);
 {obs({}) * lock($l$) $\wedge$ I($l$)=inv($b$)}
 fork( {obs({}) * lock($l$)}
  while (true) fork(reader($b$)));
 {obs({}) * lock($l$)}
 while (true) fork(writer($b$))
 {obs({}) * lock($l$)}}

routine reader(rdwr $b$){
 {obs($O$) * lock($b.l$) $\wedge\ b.l \preccurlyeq O \uplus \{b.v_w\}$
 $\wedge\ b.v_r \preccurlyeq O \wedge$ I($b.l$)=inv($b$)}
 acquire($b.l$);
 while($b.aw$+$b.ww$>0)
   wait($b.v_r, b.l$);
 $b.ar$:=$b.ar$+1;
 g_chrg($b.v_w$);
 release($b.l$);
 // Perform reading ...
 acquire($b.l$);
 if($b.ar$<1)
   abort;
 $b.ar$:=$b.ar$−1;
 if ($Wt(b.v_w) > 0$) g_inc($b.c_w$);
 notify($b.v_w$);
 g_disch($b.v_w$);
 release($b.l$) {obs({}) * lock($b.l$)}}

routine writer(rdwr $b$){
 {obs($O$) * lock($b.l$) $\wedge\ b.l \preccurlyeq O \uplus \{b.v_w, b.v_r\}$
 $\wedge\ b.v_w \preccurlyeq O \uplus \{b.v_w, b.v_r\}$ $\wedge$ I($b.l$)=inv($b$)}
 acquire($b.l$);
 g_chrg($b.v_w$); g_inc($b.c_w$);
 g_chrg($b.v_r$);
 while($b.aw$+$b.ar$>0){
   g_dec($b.c_w$);
   $b.ww$:=$b.ww$+1;
   wait($b.v_w, b.l$);
   if($b.ww$<1)
      abort();
   $b.ww$:=$b.ww$−1
 };
 $b.aw$:=$b.aw$+1;
 g_dec($b.c_w$);
 release($b.l$);
 // Perform writing ...
 acquire($b.l$);
 if($b.aw \neq 1$)
   abort;
 $b.aw$:=$b.aw$−1;
 if ($Wt(b.v_w) > 0$) g_inc($b.c_w$);
 notify($b.v_w$);
 if($b.ww$=0)
   notifyAll($b.v_r$);
 g_disch($b.v_w$); g_disch($b.v_r$);
 release($b.l$) {obs({}) * lock($b.l$)}}
```

Fig. 11. Verification of the program in Figure 10

former for the command $\mathsf{wait}(v_w, l)$ and fulfils the latter at the end of its execution. Accordingly, since when the command $\mathsf{wait}(v_w, l)$ is executed v_w is in the bag of obligations of the writer thread, this command can be verified if $v_w \preccurlyeq \{\!\!\{v_w\}\!\!\}$, where $\mathsf{P}(v_w)$ must be true. The verification of this program is illustrated in Figure 11. Generally, for a condition variable v for which $P(v)=\mathsf{true}$ a lock invariant can imply the invariant $\mathsf{one\_ob}(v, Wt, Ot)$ if it asserts $Wt(v)+Ct(v)<Ot(v)+S(v)$ and $Wt(v)=0 \vee Wt(v)<Ot(v)$, where $Ct(v)$ is the total number of credits for v and $S(v)$ is an integer value such that $\mathsf{wait}(v, l)$ is executed only if $S(v)\leqslant 0$.

4.2 A Further Relaxation

The relation $\preccurlyeq$ allows one to verify some deadlock-free programs where a thread waits for a condition variable while that thread is also obliged to fulfil an obligation for that variable. However, it is still possible to have a more general, more relaxed definition for this relation. Under this definition a thread with obligations O is allowed to wait for a condition variable v if either $v \prec O$, or there exists an obligation o such that 1) $v \prec O - \{\!\!\{o\}\!\!\}$, and 2) o satisfies the property that in any execution state, if a thread is waiting for o then there exists a thread that is not waiting for any waitable object whose wait level is equal to/greater than the wait levels of v and o. This new definition still guarantees that in any state of the execution if the program has some threads suspended, waiting for some obligations, there is always a thread obliged to fulfil the obligation o_{min} that is not suspended, where o_{min} has a minimal wait level among all waitable objects for which a thread is waiting. To satisfy the condition number 2 we introduce a new definition for $\preccurlyeq$, shown in Definition 3, that uses a new function X mapping each lock to a set of wait levels. This definition will be sound only if the proof rules ensure that for any condition variable v whose wait level is in $\mathsf{X}(\mathsf{L}(\mathsf{v}))$ the number of obligations is equal to or greater than the number of the waiting threads.

This definition is still sound because of Lemma 1, that has been machine-checked in Coq [5], where G is a bag of waitable object-bag of obligations pairs such that each element t of G is associated with a thread in a state of the execution, where the first element of t is the object for which t is waiting and the second element is the bag of obligations of t. This lemma implies that if all the mentioned rules, denoted by H_1 to H_4, are respected in any state of the execution then it is impossible that all threads in that state are waiting for a waitable object. This lemma can be proved by induction on the number of elements of G and considering the element waiting for an object whose wait level is minimal (see [16] representing its proof in details).

Definition 3 (Relaxed precedence relation). *The new precedence relation indexed over functions* $\mathsf{R}, \mathsf{L}, \mathsf{P}, \mathsf{X}$ *holds for a waitable object* v *and a bag of obligations* O*, denoted by* $v \preccurlyeq O$*, if and only if:*

[5] The machine-checked proof can be found at `https://github.com/jafarhamin/deadlock-free-monitors-soundness`

$$
\begin{array}{l}
(v{\prec}O \ \vee\ v{\preceq}O) \wedge (\neg\mathsf{exc}(v) \vee v{\perp}O)\ ,\ \textit{where}\\
v{\prec}O \Leftrightarrow \forall o{\in}O.\ \mathsf{R}(v){<}\mathsf{R}(o)\\
v{\preceq}O \Leftrightarrow \mathsf{P}(v){=}\mathsf{true} \wedge \mathsf{exc}(v) \wedge\\
\qquad\quad \exists o.\ v{\prec}O{-}\{\!\!\{o\}\!\!\} \wedge \mathsf{R}(v){\leqslant}\mathsf{R}(o){+}1 \wedge \mathsf{L}(v){=}\mathsf{L}(o) \wedge \mathsf{exc}(o)\\
\mathsf{exc}(v) = \mathsf{R}(v) \in \mathsf{X}(\mathsf{L}(v))\\
v{\perp}O \Leftrightarrow \mathsf{let}\ Ox = \lambda v'.\ \begin{cases} O(v') & \textit{if}\ \mathsf{R}(v') \in \mathsf{X}(\mathsf{L}(v))\\ 0 & \textit{otherwise}\end{cases}\ \ \mathsf{in}\\
\qquad\quad |Ox|{\leqslant}1 \wedge\\
\qquad\quad \forall v'.\ Ox(v'){>}0 \Rightarrow \mathsf{L}(v') = \mathsf{L}(v)
\end{array}
$$

Lemma 1 (A Valid Graph Is Not Deadlocked).

$$
\begin{array}{l}
\forall\ G{:}\mathit{Bags}(\mathit{WaitObjs} \times \mathit{Bags}(\mathit{WaitObjs})),\ R{:}\mathit{WaitObjs}{\rightarrow}\mathit{WaitLevels},\\
L{:}\mathit{WaitObjs}{\rightarrow}\mathit{Locks},\ P{:}\mathit{WaitObjs}{\rightarrow}\mathit{Bools},\ X{:}\mathit{Locks}{\rightarrow}\mathit{Sets}(\mathit{WaitLevels}).\\
H_1 \wedge H_2 \wedge H_3 \wedge H_4 \Rightarrow G{=}\{\!\!\{\}\!\!\}\ ,\ \textit{where}\\
\quad H_1 : \forall(o,O){\in}G.\ 0 < \mathsf{Ot}(o)\\
\quad H_2 : \forall(o,O){\in}G.\ P(o){=}\mathsf{true} \Rightarrow \mathsf{Wt}(o) < \mathsf{Ot}(o)\\
\quad H_3 : \forall(o,O){\in}G.\ R(o) \in X(L(o)) \Rightarrow \mathsf{Wt}(o) \leqslant \mathsf{Ot}(o)\\
\quad H_4 : \forall(o,O){\in}G.\ o{\preccurlyeq}_{R,L,P,X}O\\
\textit{where}\ \mathsf{Wt} = \biguplus_{(o,O)\in G} \{\!\!\{o\}\!\!\}\ \textit{and}\ \mathsf{Ot} = \biguplus_{(o,O)\in G} O
\end{array}
$$

$$\textsc{NewLock}\ \{\mathsf{true}\}\ \mathsf{newlock}\ \{\lambda l.\ \mathsf{ulock}(l, \{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}) \wedge \mathsf{R}(l){=}z \wedge \mathsf{X}(l){=}X\}$$

$$\textsc{NewCv}\ \{\mathsf{true}\}\ \mathsf{newcond}\ \{\lambda v.\ \mathsf{R}(v){=}z \wedge \mathsf{L}(v){=}l \wedge \mathsf{M}(v){=}m \wedge \mathsf{P}(v){=}b\}$$

Fig. 12. New proof rules initializing functions X and P used in safe_obs and $\preccurlyeq$

To extend the proof rules with the new precedence relation it suffices to include a new invariant own_ob in the definition of safe_obs, as shown in Definition 4, an invariant that must hold when a command wait or a ghost command g_disch is executed, to make sure that for any condition variable for which exc holds, the number of obligations is equal to/greater than the number of the waiting threads. Additionally, the functions X and P, as indicated in Figure 12, are initialized when a lock and a condition variable is created, respectively. The rest of the proof rules are the same as those defined in Figure 5 except that the old precedence relation ($\prec$) is replaced by the new one ($\preccurlyeq$).

Definition 4 (Safe Obligations). *The relation* safe_obs(v, Wt, Ot)*, indexed over functions* R, L, P, X*, holds if and only if:*

$$
\begin{array}{l}
\mathsf{one\_ob}(v,\ Wt, Ot)\ \wedge\ (\mathsf{P}(v){=}\mathsf{true} \Rightarrow \mathsf{spare\_ob}(v,\ Wt, Ot)) \wedge\\
(\mathsf{exc}(v){=}\mathsf{true} \Rightarrow \mathsf{own\_ob}(v,\ Wt, Ot)),\ \textit{where}\\
\mathsf{one\_ob}(v,\ Wt, Ot) \Leftrightarrow (\ Wt(v){>}0 \Rightarrow Ot(v){>}0)\\
\mathsf{spare\_ob}(v,\ Wt, Ot) \Leftrightarrow (\ Wt(v){>}0 \Rightarrow\ Wt(v){<}Ot(v))\\
\mathsf{own\_ob}(v,\ Wt, Ot) \Leftrightarrow (\ Wt(v){\leqslant}Ot(v))
\end{array}
$$

```
routine main(){
  $q$ := newqueue;
  $l$ := newlock;
  $v_f$ := newcvar;
  $v_e$ := newcvar;
  $ch$:=channel$(q, l, v_f, v_e)$;
  fork (receive($ch$));
  send($ch$, 12)}
```

```
routine send(channel $ch$, int $d$)
{
  acquire($ch.l$);
  while(sizeof($ch.q$) = max)
    wait($ch.v_f, ch.l$);
  enqueue($ch.q, d$);
  notify($ch.v_e$);
  release($ch.l$)}
```

```
routine receive(channel $ch$)
{
  acquire($ch.l$);
  while(sizeof($ch.q$) = 0)
    wait($ch.v_e, ch.l$);
  dequeue($ch.q$);
  notify($ch.v_f$);
  release($ch.l$)}
```

inv(channel ch) ::= $\lambda Wt.\ \lambda Ot.\ \exists Cte, Ctf.$ ctr$(ch.c_e, Cte) *$ ctr$(ch.c_f, Ctf) *$
$\exists s.$ queue$(ch.q, s)\ \wedge$ P(v_e)=false $\wedge$ M(v_e)=tic$(ch.c_e) \wedge$ M(v_f)=tic$(ch.c_f)$ *land*
L$(ch.v_e)$=L$(ch.v_f)$=$ch.l \wedge$
$Wt(ch.v_e) + Cte \leqslant Ot(ch.v_e) + s \wedge Wt(ch.v_e) \leqslant Ot(ch.v_e) \wedge$
$Wt(ch.v_f) + Ctf + s < Ot(ch.v_f) +$ max $\wedge\ (Wt(v_f) = 0 \vee Wt(ch.v_f) < Ot(ch.v_f))$

```
routine main(){
  $q$ := newqueue;
  $l$ := newlock;
  $v_f$ := newcvar;
  $v_e$ := newcvar;
  $ch$:=channel$(q, l, v_f, v_e)$;
  $ch.c_e$:=g_newctr;
  $ch.c_f$:=g_newctr;
  g_inc($ch.c_e$);
  g_inc($ch.c_f$);
  g_chrg($v_e$); g_chrg($v_f$);
  g_initl($l$);
  {obs($\{v_e, v_f\}$) $*$ lock($l$) $*$
  tic($ch.c_e$) $*$ tic($ch.c_f$) $*$
  L$(v_f)$=$l \wedge$ L$(v_e)$=$l \wedge$
  M$(v_e)$=tic$(ch.c_e) \wedge$
  M$(v_f)$=tic$(ch.c_f) \wedge$
  P$(v_f)$=true $\wedge$
  P$(v_e)$=false $\wedge$
  R$(l)$=0 $\wedge$
  R$(v_e)$=1 $\wedge$ R$(v_f)$=2 $\wedge$
  X$(l)$=$\{1, 2\} \wedge$ I$(l)$=inv}
  fork (receive($ch$));
  send($ch$, 12) {obs({})}}
```

```
routine send(channel $ch$, int $d$)
{{obs($O \uplus \{ch.v_e\}$) $*$ tic($ch.c_f$) $*$
 lock($ch.l$) $\wedge\ ch.l \preccurlyeq O \uplus \{ch.v_e\} \wedge$
 $ch.v_f \preccurlyeq O \uplus \{ch.v_e\} \wedge$I$(ch.l)$=inv}
 acquire($ch.l$);
 while(sizeof($ch.q$) = max){
   g_dec($ch.c_f$);
   wait($ch.v_f, ch.l$)};
 enqueue($ch.q, d$);
 if ( $Wt(b.v_e) > 0$)
   g_inc($b.c_e$);
 notify($ch.v_e$);
 g_disch($ch.v_e$);
 g_dec($ch.c_f$);
 release($ch.l$)
 {obs($O$) $*$ lock($ch.l$)}}
```

```
routine receive(channel $ch$){
 {obs($O \uplus \{ch.v_f\}$) $*$ tic($ch.c_e$) $*$
 lock($ch.l$) $\wedge\ ch.l \preccurlyeq O \uplus \{ch.v_f\} \wedge$
 $ch.v_e \preccurlyeq O \uplus \{ch.v_f\} \wedge$I$(ch.l)$=inv}
 acquire($ch.l$);
 while(sizeof($ch.q$) = 0){
   g_dec($ch.c_e$);
   wait($ch.v_e, ch.l$)};
 dequeue($ch.q$);
 if ( $Wt(b.v_f) > 0$)
   g_inc($b.c_f$);
 notify($ch.v_f$);
 g_disch($ch.v_f$);
 g_dec($ch.c_e$);
 release($ch.l$)
 {obs($O$) $*$ lock($ch.l$)}}
```

Fig. 13. Verification of a bounded channel synchronized using a monitor consisting of condition variables v_f, preventing sending on a full channel, and v_e, preventing taking messages from an empty channel

Bounded Channels. One application of the new definition is a bounded channel program, shown in Figure 13, where a sender thread waits for a receiver thread if the channel is full, synchronized by v_f, and a receiver thread waits for a sender thread if the channel is empty, synchronized by v_e. More precisely, the sender thread with an obligation for v_e might execute the command $\mathsf{wait}(v_f, l)$, and the receiver thread with an obligation for v_f might execute a command $\mathsf{wait}(v_e, l)$. Since v_e and v_f are not equal, it is impossible to verify this program by the old definition of $\preccurlyeq$ because the waiting levels of v_e and v_f cannot be lower than each other. Thanks to the new definition of $\preccurlyeq$, this program can be verified, as shown in Figure 13, by initializing $\mathsf{P}(v_f)$ with true and $\mathsf{X}(l)$ with $\{1, 2\}$, where two consecutive numbers 1 and 2 are the wait levels of v_e and v_f, respectively.

5 Soundness Proof

In this section we provide a soundness proof for the present approach [6], i.e. if a program is verified by the proposed proof rules, where the verification starts from an empty bag of obligations and also ends with such bag, this program is deadlock-free. To this end, we first define the syntax of programs and a small-step semantics for programs ($\rightsquigarrow$) relating two *configurations* (see [16] for formal definitions). A configuration is a thread table-heap pair (t, h), where heaps and thread tables are some partial functions from locations and thread identifiers to integers and command-*context* pairs $(c; \xi)$, respectively, where a context, denoted by ξ, is either done or $\mathsf{let}\ x{:=}[]\ \mathsf{in}\ c; \xi$. Then we define *validity of configurations*, shown in Definition 5, and prove that 1) if a program c is verified by the proposed proof rules, where it starts from the precondition $\mathsf{obs}(\{\!\!\}\!\!\})$ and satisfies the post condition $\lambda\_.\mathsf{obs}(\{\!\!\}\!\!\})$, then the initial configuration, where the heap is empty, denoted by $\mathbf{0}{=}\lambda\_.\varnothing$, and there is only one thread with command c and context done, is a valid configuration (Theorem 4), 2) a valid configuration is not deadlocked (Theorem 5), and 3) starting from a valid configuration, all the subsequent configurations of the execution are also valid (Theorem 6).

In a valid configuration (t, h), h contains all the heap ownerships that are in possession of all threads in t and also those that are in possession of the locks that are not held, specified by a list A. Additionally, each thread must have all the required permissions to be successfully verified with no remaining obligation, enforced by wpcx. $\mathsf{wpcx}(c, \xi)$ in this definition is a function returning the weakest precondition of the command c with the context ξ w.r.t. the postcondition $\lambda\_.\mathsf{obs}(\{\!\!\}\!\!\})$ (see [16] for formal definitions). This function is defined with the help of a function $\mathsf{wp}(c, a)$ returning the weakest precondition the command c w.r.t. the postcondition a.

Definition 5 (Validity of Configurations). *A configuration is valid, denoted by* $\mathsf{valid}(t, h)$*, if there exist a list of* augmented threads T*, consisting of an*

[6] The machine-checked version of some lemmas and theorems in this proof, such as Theorems 4 and 5, can be found at `https://github.com/jafarhamin/deadlock-free-monitors-soundness`.

identifier (id), a program (c), a context (ξ), a permission heap (p), a ghost resource heap (C) and a bag of obligations (O) associated with each thread; a list of assertions A, and some functions R, I, L, M, P, X such that:

– $\forall id, c, \xi.\ t(id){=}(c;\xi) \Leftrightarrow \exists p, O, C.\ (id, c, \xi, p, O, C) \in T$
– $h = \mathsf{pheap2heap}(\mathop{\ast}\limits_{a\in A} a \ast \mathop{\ast}\limits_{(id,c,\xi,p,O,C)\in T} p)$
– $\forall (id, c, \xi, p, O, C){\in}T.$
  - $p, O, C \models \mathsf{wpcx}_{R,I,L,M,P,X}(c, \xi)$
  - $\forall l,\ Wt, Ot.\ p(l){=}\mathsf{Ulock/Locked}(Wt, Ot) \Rightarrow Wt{=}\mathsf{Wt}_l \wedge Ot{=}\mathsf{Ot}_l$
  - $\forall l.\ p(l){=}\mathsf{Lock} \wedge h(l){=}1 \Rightarrow \mathsf{I}(l)(\mathsf{Wt}_l, \mathsf{Ot}_l) \in A$
  - $\forall l.\ p(l){=}\mathsf{Lock}{\vee}p(l){=}\mathsf{Locked}(\mathsf{Wt}_l, \mathsf{Ot}_l){\Rightarrow}\neg P(l){\wedge}\neg\mathsf{exc}(l){\wedge}(h(l){=}0{\Rightarrow}l{\in}\mathsf{Ot})$
  - $\forall o.\ \mathsf{waiting\_for}(c, h){=}o \Rightarrow \mathsf{safe\_obs}_{R,L,P,X}(o, \mathsf{Wt}, \mathsf{Ot})$

  where
  - $\mathsf{Ot} = \biguplus\limits_{(id,c,\xi,p,O,C)\in T} O, \mathsf{Wt} = \biguplus\limits_{(id,c,\xi,p,O,C)\in T\wedge \mathsf{waiting\_for}(c,h)=o} \{\!\![o]\!\!\}$
  - O_l *is a bag that given an object o returns* $O(o)$ *if* $L(o){=}l$ *and* 0 *if* $L(o){\neq}l$
  - $\mathsf{waiting\_for}(c, h)$ *returns the object for which c is waiting, if any*
  - $\mathsf{pheap2heap}(p)$ *returns the heap corresponding with permission heap p*

We finally prove that for each proof rule $\{a\}\ c\ \{a'\}$ we have $a \Rightarrow \mathsf{wp}(c, a')$. To this end, we first define *correctness of commands*, shown in Definition 6, and then for each proof rule $\{a\}\ c\ \{a'\}$ we prove $\mathsf{correct}(a, c, a')$. In addition to the proof rules presented in this paper, other useful rules such as the rules *consequence*, *frame* and *sequential*, shown in Theorems 1, 2, and 3 can also be proved with the help of some auxiliary lemmas in [16]. Note that the indexes R, I, L, M, P, X are omitted when they are unimportant.

Definition 6 (Correctness of Commands).
$\mathsf{correct}_{R,I,L,M,P,X}(a, c, a') \Leftrightarrow (a \Rightarrow \mathsf{wp}_{R,I,L,M,P,X}(c, a'))$

Theorem 1 (Rule Consequence).
$\mathsf{correct}(a_1, c, a_2) \wedge (a'_1 \Rightarrow a_1) \wedge (\forall z.\ a_2(z) \Rightarrow a'_2(z)) \Rightarrow \mathsf{correct}(a'_1, c, a'_2)$

Theorem 2 (Rule Frame).
$\mathsf{correct}(a, c, a') \Rightarrow \mathsf{correct}(a \ast f, c, \lambda z.\ a'(z) \ast f)$

Theorem 3 (Rule Sequential Composition).
$\mathsf{correct}(a, c_1, a') \wedge (\forall z.\ \mathsf{correct}(a'(z), c_2[z/x], a'')) \Rightarrow$
$\mathsf{correct}(a, \mathsf{let}\ x{:=}c_1\ \mathsf{in}\ c_2, a'')$

Theorem 4 (The Initial Configuration Is Valid).
$\mathsf{correct}_{R,I,L,M,P,X}(\mathsf{obs}(\{\!\![]\!\!\}), c, \lambda\_.\mathsf{obs}(\{\!\![]\!\!\})) \Rightarrow \mathsf{valid}(\mathbf{0}[id{:=}c; \mathsf{done}], \mathbf{0})$

Proof. The goal is achieved because there are an augmented thread list $T{=}[(id, c, \mathsf{done}, \mathbf{0}, \{\!\![]\!\!\}, \mathbf{0})]$, a list of assertions $A{=}[]$, and functions R, I, L, M, P, X by which all the conditions in the definition of validity of configurations are satisfied.

Theorem 5 (A Valid Configuration Is Not Deadlocked).
$(\exists id, c, \xi, o.\ t(id){=}(c;\xi) \wedge \mathsf{waiting\_for}(c, h){=}o) \wedge \mathsf{valid}(t, h)$
$\Rightarrow \exists id', c', \xi',\ t(id'){=}(c';\xi') \wedge \mathsf{waiting\_for}(c', h){=}\varnothing$

Proof. We assume that all threads in t are waiting for an object. Since (t, h) is a valid configuration there exists a valid augmented thread table T with a corresponding valid graph $G{=}\mathsf{g}(T)$, where g maps any element such as (id, c, ξ, p, O, C) to a new one such as $(\mathsf{waiting\_for}(c), O)$. By Lemma 1, we have $G{=}\{\!\!\}$, implying $T{=}\{\!\!\}$, implying $t{=}\mathbf{0}$ which contradicts the assumption of the theorem.

Theorem 6 (Steps Preserve Validity of Configurations). [7]
$\mathsf{valid}(\kappa) \wedge \kappa \rightsquigarrow \kappa' \Rightarrow \mathsf{valid}(\kappa')$

Proof. By case analysis of the small step relation $\rightsquigarrow$ (see [16] explaining the proof of some non-trivial cases).

6 Related Work

Several approaches to verify termination [1,21], total correctness [3], and lock freedom [2] of concurrent programs have been proposed. These approaches are only applicable to non-blocking algorithms, where the suspension of one thread cannot lead to the suspension of other threads. Consequently, they cannot be used to verify deadlock-freedom of programs using condition variables, where the suspension of a notifying thread might lead a waiting thread to be infinitely blocked. In [22] a compositional approach to verify termination of multi-threaded programs is introduced, where *rely-guarantee reasoning* is used to reason about each thread individually while there are some assertions about other threads. In this approach a program is considered to be terminating if it does not have any infinite computations. As a consequence, it is not applicable to programs using condition variables because a waiting thread that is never notified cannot be considered as a terminating thread.

There are also some other approaches addressing some common synchronization bugs of programs in the presence of condition variables. In [8], for example, an approach to identify some potential problems of concurrent programs consisting waits and notifies commands is presented. However, it does not take the order of execution of theses commands into account. In other words, it might accept an undesired execution trace where the waiting thread is scheduled before the notifying thread, that might lead the waiting thread to be infinitely suspended. [9] uses Petri nets to identify some common problems in multithreaded programs such as data races, lost signals, and deadlocks. However the model introduced for condition variables in this approach only covers the communication of two threads and it is not clear how it deals with programs having more than two threads communicating through condition variables. Recently, [10] has introduced an approach ensuring that every thread synchronizing under a set of condition variables eventually exits the synchronization block if that thread eventually reaches that block. This approach succeeds in verifying one of the applications of condition variables, namely the buffer. However, since this approach is not modular and relies on a Petri net analysis tool to solve the termination

[7] The proof of this theorem has not been machine-checked with Coq yet.

problem, it suffers from a long verification time when the size of the state space is increased, such that the verification of a buffer application having 20 producer and 18 consumer threads, for example, takes more than two minutes.

Kobayashi [20,6] proposed a type system for deadlock-free processes, ensuring that a well-typed process that is annotated with a finite *capability level* is deadlock free. He extended channel types with the notion of *usages*, describing how often and in which order a channel is used for input and output. For example, usage of x in the process $x?y|x!1|x!2$, where $?, !, |$ represent an input action, an output action, and parallel composition receptively, is expressed by $?|!|!$, which means that x is used once for input and twice for output possibly in parallel. Additionally, to avoid circular dependency each action α is associated with the levels of obligation o and capabilities c, denoted by α^o_c, such that 1) an obligation of level n must be fulfilled by using only capabilities of level less than n, and 2) for an action of capability level n, there must exist a co-action of obligation level less than or equal to n. Leino *et al.* [4] also proposed an approach to verify deadlock-freedom of channels and locks. In this approach each thread trying to receive a message from a channel must spend one credit for that channel, where a credit for a channel is obtained if a thread is obliged to fulfil an obligation for that channel. A thread can fulfil an obligation for a channel if either it sends a message on that channel or delegate that obligation to other thread. The same idea is also used to verify deadlock-freedom of semaphores [7], where acquiring (i.e. decreasing) a semaphore consumes one credit and releasing (i.e. increasing) that semaphore produces one credit for that semaphore. However, as it is acknowledged in [4], it is impossible to treat channels (and also semaphores) like condition variables; a wait cannot be treated like a receive and a notify cannot be treated like a send because a notification for a condition variable will be lost if no thread is waiting for that variable. We borrow many ideas, including the notion of obligations/credits(capabilities) and levels, from these works and also the one introduced in [11], where a corresponding separation logic based approach is presented to verify total correctness of programs in the presence of channels.

7 Conclusion

It this article we introduced a modular approach to verify deadlock-freedom of monitors. We also introduced a relax, more general precedence relation to avoid cycles in the wait-for graph of programs, allowing a verification approach to verify a wider range of deadlock-free programs in the presence of monitors, channels and other synchronization mechanisms.

8 Acknowledgements

This work was funded through Flemish Research Fund grant G.0058.13 and KU Leuven Research Fund grant OT/13/065. We thank three anonymous reviewers and Prof. Aleksandar Nanevski for their careful reading of our manuscript and their many insightful comments and suggestions.

References

1. Liang, H., Feng, X., Shao, Z.: Compositional verification of termination-preserving refinement of concurrent programs. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), ACM (2014) 65
2. Hoffmann, J., Marmar, M., Shao, Z.: Quantitative reasoning for proving lock-freedom. In: Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on, IEEE (2013) 124–133
3. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P., Sutherland, J.: Modular termination verification for non-blocking concurrency. In: ESOP. (2016) 176–201
4. Leino, K.R.M., Müller, P., Smans, J.: Deadlock-free channels and locks. In: European Symposium on Programming, Springer (2010) 407–426
5. Boström, P., Müller, P.: Modular verification of finite blocking in non-terminating programs. Volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
6. Kobayashi, N.: A new type system for deadlock-free processes. In: CONCUR. Volume 6., Springer (2006) 233–247
7. Jacobs, B.: Provably live exception handling. In: Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, ACM (2015) 7
8. Wang, C., Hoang, K.: Precisely deciding control state reachability in concurrent traces with limited observability. In: VMCAI, Springer (2014) 376–394
9. Kavi, K.M., Moshtaghi, A., Chen, D.J.: Modeling multithreaded applications using petri nets. International Journal of Parallel Programming **30**(5) (2002) 353–371
10. de Carvalho Gomes, P., Gurov, D., Huisman, M.: Specification and verification of synchronization with condition variables. In: International Workshop on Formal Techniques for Safety-Critical Systems, Springer (2016) 3–19
11. Jacobs, B., Bosnacki, D., Kuiper, R.: Modular termination verification. In: LIPIcs-Leibniz International Proceedings in Informatics. Volume 37., Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
12. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for c and java. NASA Formal Methods **6617** (2011) 41–55
13. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. Programming Languages and Systems (2010) 304–311
14. Jacobs, B., ed.: VeriFast 18.02. Zenodo, `http://doi.org/10.5281/zenodo.1182724`. (2018)
15. Dijkstra, E.W.: Cooperating sequential processes. In: The origin of concurrent programming. Springer (1968) 65–138
16. Hamin, J., Jacobs, B.: Deadlock-free monitors: extended version. Technical report CW712, Department of Computer Science, KU Leuven, Belgium (2018)
17. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. ACM SIGPLAN Notices **46**(1) (2011) 271–282
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on, IEEE (2002) 55–74
19. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. ACM SIGPLAN Notices **50**(1) (2015) 637–650

20. Kobayashi, N.: Type systems for concurrent programs. In: Formal Methods at the Crossroads. From Panacea to Foundational Support. Springer (2003) 439–453
21. Hamin, J., Jacobs, B.: Modular verification of termination and execution time bounds using separation logic. In: Information Reuse and Integration (IRI), 2016 IEEE 17th International Conference on, IEEE (2016) 110–117
22. Popeea, C., Rybalchenko, A.: Compositional termination proofs for multi-threaded programs. In: TACAS. Volume 12., Springer (2012) 237–251
23. Vafeiadis, V.: Concurrent separation logic and operational semantics. Electronic Notes in Theoretical Computer Science **276** (2011) 335–351