Starvation-Free Monitors^{*}

Jafar Hamin^[0000–0002–5701–9111]

imec-DistriNet, Department of Computer Science, KU Leuven, Belgium
`jafar.hamin@cs.kuleuven.be`

Abstract. Monitors are a synchronization construct which allows to keep a thread waiting until a specific resource for that thread is available. One potential problem with these constructs is starvation; a situation where a thread, competing for a resource, infinitely waits for that resource because other threads, that started competing for that resource later, get it earlier infinitely often. In this paper a modular approach to verify starvation-freedom of monitors is presented, ensuring that each time that a resource is released and its associated condition variable is notified each waiting thread approaches the front of the waiting queue; more specifically, the loop in which the wait command is executed (that checks the waiting condition) has a loop variant. To this end, we introduce notions of *publishable resources* and *publishable obligations*, which are published from the thread notifying a condition variable to all of the threads waiting for that condition variable. The publishable resources ensure the waiting threads that they are approaching the front of the waiting queue, by allowing to define an appropriate loop variant for the related loop. The publishable obligations ensure that for any thread waiting for a condition variable v there is another thread obliged to notify v , which only waits for waitable objects whose levels, some arbitrary numbers associated with each waitable object, are lower than the level of v (preventing circular dependencies). We encoded the proposed separation logic-based proof rules in the VeriFast program verifier and succeeded in verifying deadlock-freedom and starvation-freedom of two monitors, having no scheduling policy, which implement two common queue locking algorithms, namely *ticket lock* and *CLH lock*.

Keywords: Starvation-freedom · Monitors · Condition variables · Modular verification · Separation logic · Hoare logic.

1 Introduction

Multithreaded programs use synchronizations such as locks and condition variables to control the execution of threads. Monitors are one of the popular constructs used to synchronize threads [34, 32, 33]. This construct, consisting of a

^{*} We thank three anonymous reviewers and also Prof. Bart Jacobs for their careful reading of our manuscript and their many insightful comments and suggestions. This research received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 731453 (project VESSEDIA).

lock and some condition variables associated with that lock, provides some APIs for its clients, namely `wait( $v, l$ )`, causing the calling thread to release the lock l and wait for a notification on the condition variable v , and `notify( $v$ )`/`notifyAll( $v$ )`, causing one/all thread(s) waiting for v to resume its/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 before it can continue execution.

Monitors allow to keep a thread waiting until a desired condition is met, e.g. an item is produced (in a producer-consumer program), a writer finishes its writing (in a readers-writer lock), or a specific number of threads finish a specific task (in a barrier). However, one potential problem with these constructs is starvation; a situation where a thread, competing for a resource, infinitely waits for that resource because other threads, that started competing for that resource later, get it earlier infinitely often. For example, a shared buffer allows starvation if it is possible that a thread starts a *take* operation but it never completes, while other threads' *take* operations complete infinitely often. As another example, a readers-writers lock allows starvation if it is possible that a thread attempts to enter the write lock but never succeeds, while other threads successfully acquire and release the write lock infinitely often.

Several approaches to verify termination, deadlock-freedom, liveness, and finite blocking of threads have been introduced. Some of these approaches only work with non-blocking algorithms [27], 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 notifying thread causes other waiting threads to be blocked forever. Some other approaches are also presented to verify termination of programs using some blocking constructs such as channels [20, 19, 13] and semaphores [11]. These approaches are not general enough to cover condition variables because unlike channels and semaphores a notification on a condition variable is lost when there is no thread waiting for that condition variable. There are also some approaches [35, 17, 4] to verify correctness of programs in the presence of monitors. However, these approaches either only cover a very specific application of monitors, 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.

Recently, Hamin *et al.* [6, 7] introduced a modular approach to verify absence of deadlock in monitors. This approach ensures that in any state of the execution there is a thread running, not waiting for a lock or condition variable, 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 discharge an obligation for v that only waits for objects whose levels, some arbitrary numbers associated with each waitable object, are lower than the level of v . The notion of obligations in this approach, which is borrowed from the approach of [20], verifying deadlock-freedom of channels, and Kobayashi's type system [19], verifying deadlock-freedom of π -calculus processes, makes it

```

routine new_tvm()  routine enter(tvm t)  routine leave(tvm t)  routine main(){
{
  l := new_lock();
  v := new_cvar();
  tvn(l:=l, v:=v,
  active:=new_int(0),
  next:=new_int(0))
}

{
  acquire(t.l);
  mine := t.next++;
  while(t.active < mine)
    wait(t.v, t.l);
  release(t.l)
}

{
  acquire(t.l);
  t.active++;
  notifyAll(t.v);
  release(t.l)
}

t:=new_tvm;
fork(while(1){
  enter(t); // CS
  leave(t)
});
enter(t); // CS
leave(t)}

```

Fig. 1. A monitor implementing a ticket algorithm.

possible that different modules (routines) of a program are individually verified. However, in this approach a thread which holds a resource and has an obligation to release that resource cannot transfer that obligation to the next thread, which acquires that resource. Later, this problem is solved in [9] by allowing obligations to be transferred from the notifying thread to the one notified. In this approach the transferred obligations are discharged from the notifying thread only if there is a waiting thread onto which the transferred obligations are loaded. However, in this approach obligations can be transferred using `notify(v)` but not using `notifyAll(v)`, which wakes up all the threads waiting for the condition variable v .

This constraint does not allow this approach to verify a monitor implementing a ticket lock algorithm, shown in Figure 1, for example, where the releasing thread notifies all the waiting threads. As shown in the routine `new_tvm` in this figure, a `tvm` structure consists of a variable `active`, storing the now serving ticket number, a variable `next`, storing the last ticket number, a lock l , protecting these variables from concurrent accesses, and a condition variable v , keeping a thread waiting until its ticket number equals the now serving ticket number. As shown in the routine `enter`, the running thread first acquires the lock l , then it assigns the last ticket number to its own ticket number, denoted by `mine`, and increases the last ticket number. While the now serving ticket number is lower than the ticket number of this thread it releases the lock l and waits for a notification on the condition variable v . When this thread is notified (and reacquires l) if its ticket number is equal (or lower than) the now serving ticket number it releases l and continues its execution. After this thread gets service, as shown in the routine `leave`, it acquires l , increases the now serving ticket number, notifies all threads waiting for v , and releases l . After this notification, the thread whose ticket number equals the now serving ticket number continues its execution and the rest of the threads wait for a notification again.

Ticket lock algorithms and CLH lock algorithms [2] are two common queue locking algorithms used to enforce fairness in concurrent constructs such as a concurrent queue and a readers-writers lock (see Figure 2). To verify deadlock-

freedom and starvation-freedom¹ of monitors which have no scheduling policy² and enforce fairness by implementing these algorithms, in addition to transferring obligations using `notifyAll`, it is also necessary to transfer some information from the notifying thread to the ones notified. This information must ensure the awoken threads that they are approaching the front of the waiting queue (for example, in Figure 1 this information implies that *the now serving ticket number has been increased*). Although the approach presented in [6] allows permissions to be transferred through `notifyAll`, it considers permissions as linear resources, which cannot be in possession of more than one thread at the same time. Consequently, in that approach a notifying thread can transfer a permission p to n number of waiting threads only if it owns n instances of p .

```

routine new_rdwr(){
  l := new_lock;
  v_w := new_cvar; v_r := new_cvar;
  rdwr(l:=l, v_w:=v_w, v_r:=v_r, ar:=ar,
    active:=new_int(0), next:=new_int(0))}

routine acquire_read(rdwr b){
  acquire(b.l);
  while(b.active < b.next)
    wait(b.v_r, b.l);
  ar := ar + 1;
  release(b.l)}

routine release_read(rdwr b){
  acquire(b.l);
  ar := ar - 1;
  notifyAll(b.v_w);
  release(b.l)}

routine acquire_write(rdwr b)
{
  acquire(b.l);
  mine := b.next++;
  while(0 < b.ar ∨ b.active < mine)
    wait(b.v_w, b.l);
  release(b.l)
}

routine release_write(rdwr b)
{
  acquire(b.l);
  b.active++;
  notifyAll(b.v_w);
  if(b.active = b.next)
    notifyAll(b.v_r);
  release(b.l)
}

```

Fig. 2. A readers-writers lock which uses a ticket algorithm to make sure that no writer is starved, where ar denotes the number of the threads which are reading, the condition variable v_r keeps a reader waiting until all writers write, and the condition variable v_w keeps a writer waiting until its ticket number is equal the now serving ticket number and no other thread is reading.

In this paper we introduce a mechanism which allows obligations to be transferred from the notifying thread, executing `notifyAll`, to one of the notified threads. In this mechanism we make sure that a notified thread is successfully verified in the both cases where it receives the transferred obligations and where

¹ Note that we assume that the lock associated with monitors is a fair lock.

² Note that in a monitor having no scheduling policy the blocked thread which is awoken by `notify` is not necessarily chosen in priority order; this monitor do not keep the order in which threads start to wait.

it does not receive the transferred obligations. Additionally, we introduce a notion of publishable resources which can be published from a thread to many other threads. This allows to transfer this information that *the waiting threads are approaching the front of the waiting queue* from a notifying thread to all of the threads notified. Having this information in a notified thread, it is possible to make a loop variant for the loop in which the wait command is executed (the loop that checks the waiting condition). Note that this loop variant ensures that the thread in the monitor which is waiting for a resource is not starved, i.e. each time that this resource is released and its associated condition variable is notified this thread approaches the front of the waiting queue. We encoded the proposed proof rules in the VeriFast program verifier [14, 15, 12] and verified that the program in Figure 1 is deadlock-free and the loop in the routine `enter` is a finite loop, implying starvation-freedom of the related monitor³.

This paper is structured as follows. Section 2 provides some background information on the existing approaches that verify deadlock-freedom of monitors. Section 3 introduces a mechanism to transfer obligations through broadcasts, which makes it possible to verify deadlock-freedom of monitors implementing some queue locking algorithms, which cannot be verified without such transfer. Section 4 introduces a mechanism to transfer some publishable resources through broadcasts, which makes it possible to verify starvation-freedom of monitors implementing some queue locking algorithms. Section 5 compares this approach with some related works and a conclusion is drawn in Section 6.

2 Deadlock-Free Monitors

Hamin *et al.* [6, 7, 9] introduced a modular approach for verifying absence of deadlock in programs synchronized using condition variables (CVs), where executing a command `wait( $v, l$ )` on a CV v , which is associated with a lock l , releases l and suspends the running thread, and executing commands `notify( $v$ )` or `notifyAll( $v$ )` wake up one or all thread(s) waiting for CV v at which point they will try to reacquire l . This approach ensures absence of deadlock by making sure that for any CV v for which a thread is waiting there is a thread obliged to fulfill an obligation for v which only waits for waitable objects whose levels are lower than the level of v . In this approach when a thread acquires a lock l , the total number of waiting threads, and the total number of obligations of any CV v associated with l , denoted by $Wt(v)$ and $Ot(v)$ respectively, can be mentioned in the proof of that thread. Wt and Ot are actually two bags⁴ which keep track of the total number of the waiting threads and the total number of the obligations of the condition variables associated with a lock, respectively. In order to ensure the mentioned constraint this approach makes sure that 1) if a command `wait( $v, l$ )` is executed then $0 < Ot(v)$, i.e. there is an obligation for v

³ The proof of this program, verified by the VeriFast program verifier, and the proof of a CLH lock algorithm can be found in [8].

⁴ We model bags of objects as functions from objects to natural numbers. We also use $\uplus$ indicating the union of two bags.

$$\begin{array}{l}
\text{NEWLOCK} \\
\{\text{true}\} \text{ newlock } \{\lambda l. \text{ ulock}(l, \{\}, \{\}) \wedge R(l)=r\} \\
\\
\text{INITLOCK} \\
\{\text{ulock}(l, Wt, Ot) * \text{inv}(Wt, Ot) * \text{obs}(O)\} \text{ g_initl}(l) \{\lambda_. \text{ lock}(l) * \text{obs}(O) \wedge I(l)=\text{inv}\} \\
\\
\text{ACQUIRE} \\
\{\text{lock}(l) * \text{obs}(O) \wedge l \prec O\} \text{ acquire}(l) \\
\{\lambda_. \exists Wt, Ot. \text{ locked}(l, Wt, Ot) * I(l)(Wt, Ot) * \text{obs}(O \uplus \{l\})\} \\
\\
\text{RELEASE} \\
\{\text{locked}(l, Wt, Ot) * I(l)(Wt, Ot) * \text{obs}(O \uplus \{l\})\} \text{ release}(l) \{\lambda_. \text{ lock}(l) * \text{obs}(O)\} \\
\\
\text{NEWCV} \\
\{\text{true}\} \text{ new_cvar } \{\lambda v. \text{ ucond}(v) \wedge R(v)=r\} \\
\\
\text{INITCV} \\
\{\text{ucond}(v) * \text{ulock}(l, Wt, Ot)\} \text{ g_initc}(v) \\
\{\lambda_. \text{ cond}(v, M, M') * \text{ulock}(l, Wt, Ot) \wedge L(v) = l\} \\
\\
\text{WAIT} \\
\{\text{cond}(v, M, M') * \text{locked}(l, Wt, Ot) * I(l)(Wt \uplus \{v\}, Ot) * \text{obs}(O \uplus \{l\}) \wedge 0 < Ot(v) \wedge \\
l = L(v) \wedge v \prec O \wedge l \prec O \uplus M'\} \text{ wait}(v, l) \{\lambda_. \text{ cond}(v, M, M') * \text{obs}(O \uplus \{l\} \uplus M') * \\
\exists Wt', Ot'. \text{ locked}(l, Wt', Ot') * I(l)(Wt', Ot') * M\} \\
\\
\text{NOTIFY} \\
\{\text{obs}(O \uplus (0 < Wt(v) ? M' : \{\})) * \text{cond}(v, M, M') * \text{locked}(L(v), Wt, Ot) * \\
(Wt(v)=0 \vee M)\} \text{ notify}(v) \{\lambda_. \text{ obs}(O) * \text{cond}(v, M, M') * \text{locked}(L(v), Wt - \{v\}, Ot)\} \\
\\
\text{NOTIFYALL} \\
\{\text{cond}(v, M, \{\}) * \text{locked}(L(v), Wt, Ot) * (\prod_{i=1}^{Wt(v)} M)\} \text{ notifyAll}(v) \\
\{\lambda_. \text{ cond}(v, M, \{\}) * \text{locked}(L(v), Wt[v:=0], Ot)\} \\
\\
\text{CHARGE OBLIGATION} \\
\{\text{obs}(O) * \text{locked}(L(v), Wt, Ot)\} \text{ g_charge}(v) \\
\{\lambda_. \text{ obs}(O \uplus \{v\}) * \text{locked}(L(v), Wt, Ot \uplus \{v\})\} \\
\\
\text{DISCHARGE OBLIGATION} \\
\{\text{obs}(O) * \text{locked}(L(v), Wt, Ot) \wedge (0 < Wt(v) \Rightarrow 1 < Ot(v))\} \text{ g_discharge}(v) \\
\{\lambda_. \text{ obs}(O - \{v\}) * \text{locked}(L(v), Wt, Ot - \{v\})\}
\end{array}$$

Fig. 3. Proof rules verifying deadlock-freedom of monitors, where $\text{obs}(O)$ denote the bag of the obligations of the current thread; $R(o)$ specifies the level of the waitable object o ; $I(l)$ specifies the invariant of the lock l ; $Wt(v)$ and $Ot(v)$ in the permission $\text{ulock}/\text{locked}$ denote the total number of threads waiting for v and the total number of obligations for v , respectively; the parameters M and M' in the permission cond of a condition variable denote the permissions and the obligations which are transferred from the thread notifying that condition variable to the one notified, respectively; $L(v)$ denotes the lock associated with the condition variable v ; and $v \prec O \Leftrightarrow \forall o \in O. R(v) < R(o)$.

in the system, 2) if an obligation for v is discharged then after this discharge the invariant $0 < Wt(v) \Rightarrow 0 < Ot(v)$ holds, i.e. if there is a thread waiting for v then after this discharge there are still some obligations for v in the system, and 3) a thread executes a command $\text{wait}(v, l)$ only if the level of v is lower than the levels of the obligations of that thread.

A program in this approach can be successfully verified if each lock associated with some CVs has an appropriate invariant such that for any CV v associated with that lock this invariant implies $0 < Wt(v) \Rightarrow 0 < Ot(v)$. Accordingly, in this approach each lock invariant is parametrized over the bags Wt and Ot , which map all CVs associated with that lock to the number of their waiting threads and obligations, respectively.

The proof rules proposed in this approach are shown in Figure 3. As shown in Rule NEWLOCK, when a lock l is created an arbitrary level is assigned to that lock by the proof author, denoted by $R(l)$, and an uninitialized lock permission $\text{ulock}(l, \emptyset, \emptyset)$ is produced. The second and the third parameters of this permission are two bags mapping the CVs associated with l to the total number of threads waiting for that CV and the total number of obligations of that CV in the system, respectively. As shown in Rule INITLOCK, using a ghost command⁵ g_initl , this uninitialized lock permission can be converted to a duplicable lock permission $\text{lock}(l)$ if the assertion resulting from applying the invariant of that lock, denoted by $I(l)$, to the bags stored in the permission ulock is consumed (the permissions described by the invariant of this lock are transferred from the thread to the lock). As shown in Rule ACQUIRE, when a thread acquires this lock the permissions described by the invariant of this lock are transferred from the lock to the thread. Additionally, a permission $\text{locked}(l, Wt, Ot)$ is provided for the thread, where Wt and Ot are two bags mapping the CVs associated with l to their number of waiting threads and obligations, respectively, and are existentially quantified in the postcondition. Note that to prevent circular dependencies the precondition of this rule enforces that the level of l be lower than the levels of the obligations of the acquiring thread. Additionally, this lock is added to the bag of the obligations of this thread. As shown in rule RELEASE, when this lock is released it is discharged from the bag of the obligations and the assertion resulting from applying the invariant of this lock to the bags stored in the permission locked is consumed. Additionally, the permission locked is converted to a permission lock .

As shown in Rule NEWCV, when a CV is created an arbitrary level is assigned to it and an uninitialized permission ucond for that CV is produced. As shown in Rule INITCV, this permission can be converted to a duplicable permission cond if a lock is associated to this CV, denoted by $L(v)$. Additionally, the transferred permissions and obligations of this CV, denoted by M and M' , which are transferred from the notifying thread to the one notified, are also specified in this rule. These permissions and obligations are consumed when a command $\text{notify}(v)$ is executed (if there is a thread waiting for v ; see the pre-

⁵ The ghost commands are inserted into the program for verification purposes and have no effect on the program's behavior.

condition of Rule NOTIFY), and are produced when a command $\text{wait}(v, l)$ is executed (see the postcondition of Rule WAIT). Note that $\text{notifyAll}(v)$ transfers $Wt(v)$ instances of these permissions, denoted by $\overset{Wt(v)}{*}_{i:=1} M$ (see the precondition of Rule NOTIFYALL). Also note that $\text{notifyAll}(v)$ transfers no obligation. As shown in Rule WAIT, when a command $\text{wait}(v, l)$ is executed, since l is going to be released and the number of the threads waiting for v is going to be increased, the result of applying the invariant of lock l to bags $Wt \uplus \{v\}$ and Ot must be consumed, where Wt and Ot are the bags stored in the permission locked of l . Additionally, the level of v must be lower than the levels of all obligations of the thread except for l . Note that the level of l must be lower than the levels of these obligations and those which are transferred from the notifying thread, denoted by M' , since when the thread is woken up it tries to reacquire l . As previously mentioned, the precondition of this rule also makes sure that $0 < Ot(v)$. This precondition follows from $l(l)(Wt \uplus \{v\}, Ot)$ provided that the invariant of l is properly defined such that for any CV v' associated with l this invariant implies that if there is a thread waiting for v' there is an obligation for v' in the system. Lastly the precondition of this command makes sure that v is associated with lock l , which is enforced by $L(v)=l$. As shown in Rules NOTIFY/NOTIFYALL, when a CV v is notified, one/all instance(s) of v is/are removed from the bag Wt stored in the permission locked of the lock associated with v , if any. An obligation for a CV v is loaded/unloaded when that obligation is also loaded/unloaded onto/from the bag Ot stored in the permission locked of the lock associated with v , as shown in Rules CHARGE OBLIGATION and DISCHARGE OBLIGATION. However, an obligation for v is discharged only if after this discharge we have $0 < Wt(v) \Rightarrow 0 < Ot(v)$, which is enforced by the assertion $0 < Wt(v) \Rightarrow 1 < Ot(v)$ in the precondition of the rule DISCHARGE OBLIGATION.

3 Publishable Obligations

To verify deadlock-freedom of the program shown in Figure 1 it is necessary to transfer an obligation of v from the thread running `leave` to the thread whose ticket number equals to the new value of `active`, if there exists such a thread. However, it is impossible to verify deadlock-freedom of this program using the proof system introduced by Hamin *et al.* [6, 9] due to the following reasons: 1) this approach does not allow obligations to be transferred through broadcasts, because it is not clear which of the waiting threads should receive the transferred obligations, and 2) (even if such a transfer is possible) this approach allows obligations to be transferred through notifications only if there is a thread waiting for that condition variable ($0 < Wt(v)$), because it ensures that a thread will immediately receive the transferred obligations. Note that in this program there might be a situation where the thread whose ticket number equals the new value of `active` is not waiting for v but has been already awoken (by a broadcast issued beforehand) and is waiting to reacquire the lock associated with v .

We address the problems above by extending the Hamin *et al.* approach such that, under some circumstances, we allow obligations to be transferred through

$$\begin{array}{l}
\text{INITCV} \\
\{ \text{ucond}(v) * \text{ulock}(l, Wt, Ot, It) \} \text{g_initc}(v) \\
\{ \lambda _. \text{cond}(v, M, M') * \text{ulock}(l, Wt, Ot, It) \wedge L(v) = l \} \\
\\
\text{WAIT} \\
\{ \text{cond}(v, M, M') * \text{locked}(l, Wt, Ot, It) * l(l)(Wt \uplus \{v\}, Ot, It \cup \{(v, id)\}) * \text{obs}(O \uplus \{l\}) \wedge \\
\quad l = L(v) \wedge v \prec O \wedge l \prec O \uplus M' \wedge 0 < Ot(v) \} \text{wait}(v, l) \\
\{ \lambda _. \text{cond}(v, M, M') * M * \exists Wt', Ot', It'. l(l)(Wt', Ot', It') * \\
(v, id) \in It' ? \text{locked}(l, Wt', Ot', It' - \{(v, id)\}) * \text{obs}(O \uplus \{l\}) : \\
\quad \text{locked}(l, Wt', Ot', It') * \text{obs}(O \uplus \{l\} \uplus M') \} \\
\\
\text{NOTIFYALL} \\
\{ \text{obs}(O \uplus ((v, id) \in It ? M' : \{ \})) * \text{cond}(v, M, M') * \text{locked}(L(v), Wt, Ot, It) * \left(\begin{smallmatrix} Wt(v) \\ * \\ i:=1 \end{smallmatrix} M \right) \} \\
\text{notifyAll}(v) \{ \lambda _. \text{obs}(O) * \text{cond}(v, M, M') * \text{locked}(L(v), Wt[v:=0], Ot, It - \{(v, id)\}) \}
\end{array}$$

Fig. 4. New proof rules verifying deadlock-freedom of monitors allowing obligations to be transferred through broadcasts.

broadcasts even if there is no thread waiting for the related condition variable. To this end, in addition to the bags Wt and Ot , we introduce a new set It consisting of condition variable-identifier pairs, such that for any condition variable v and identifier id if $(v, id) \in It$ then there exists a thread such that 1) this thread either 1-1) is waiting for v , or 1-2) is awoken by a notification on v but has not yet reacquired the lock associated with v and it also has not received any obligation through that notification, and 2) if (v, id) is removed from It then the transferred obligations, which are transferred through a notification on v , are loaded onto this thread. Having this set, we allow some obligations to be transferred through broadcasts on v from the notifying thread to a thread in It , if there exists such a thread.

The updated version of the proof rules is shown in Figure 4, where, in addition to the bags Wt and Ot , the permissions $\text{locked}/\text{ulock}$ and the lock invariants are parametrized over the set It (see Rules **INITCV**). As shown in Rule **NOTIFYALL**, the transferred obligations are discharged from the notifying thread only if the set It , stored in the associated locked permission, is not empty. After this notification a thread is removed from It , if any. As shown in Rule **WAIT** a thread having a permission $\text{locked}(l, Wt, Ot, It)$ can wait for the condition variable associate with l only if there exists an identifier id such that the assertion resulting from applying the invariant of l to $Wt \uplus \{v\}$, Ot , and $It \cup \{(v, id)\}$ is consumed, because this thread is going to wait for v . When this thread is awoken the result of applying the invariant of the associated lock to the two new bags Wt' and Ot' and the new set It' is produced for the thread. If (v, id) is still in It' it means that this thread has not yet been the target of an obligation transfer, hence the transferred obligations are not loaded onto this thread. Additionally, this thread is removed from the set It' , stored in the related locked permission (because it is awoken and has acquired the associated lock). If (v, id) is not in It' anymore it

means that this thread has been the target of an obligation transfer, hence the transferred obligations must be loaded onto this thread.

<p>NEWCOUNTER $\{\text{true}\} \text{g_new_ctr} \{\lambda g. \text{ctr}(g, 0)\}$</p>	<p>INCCOUNTER $\{\text{ctr}(g, n)\} \text{g_inc}(g) \{\lambda _. \text{ctr}(g, n+1) * \text{tic}(g)\}$</p>
<p>DECCOUNTER $\{\text{ctr}(g, n) * \text{tic}(g)\} \text{g_dec}(g) \{\lambda _. \text{ctr}(g, n-1) \wedge 0 < n\}$</p>	
<p>NEWICOUNTER $\{\text{true}\} \text{g_new_ictr} \{\lambda g. \text{ictr}(g, 0)\}$</p>	<p>INCICOUNTER $\{\text{ictr}(g, n)\} \text{g_iinc}(g) \{\lambda _. \text{ictr}(g, n+1) * \text{itic}(g)\}$</p>

Fig. 5. Ghost counters and ghost monotonic counters.

It can be proved that any program verified by the mentioned proof rules, where the verification starts from an empty bag of obligations and also ends with such a bag, never deadlocks, i.e. it always has a running thread, not waiting for any waitable object such as a condition variable or a lock, until it terminates. We know that for any waitable object o all of these proof rules preserve the invariant $0 < Wt(o) \Rightarrow 0 < Ot(o)$, where $Wt(o)$ and $Ot(o)$ denote the total number of waiting threads and obligations for o in the system, respectively. Additionally, these rules ensure that a thread waits for a waitable object o only if the level of o is lower than the levels of the obligations of that thread. Note that these two invariants hold even when some obligations are transferred using `notifyAll`. Now consider a deadlocked state, where each thread of a verified program is waiting for an object. Among all of these objects take the one having a minimal wait level, namely o_{min} . By the invariants above there exists a thread having an obligation for o_{min} that is waiting for an object whose level is lower than the level of o_{min} , which contradicts minimality of the level of o_{min} .

Using the proposed proof rules it is possible to verify deadlock-freedom of the program shown in Figure 1, as shown in Figures 6 and 7. Note that to verify this program it is necessary to 1) keep track of the number of the elements in the set It , and 2) make sure that the variable `next` is not decreased at all. To meet the first condition we use the notion of *ghost counters* and corresponding *ghost counter tickets* [6], both of which are a particular kind of ghost resources⁶. Specifically, as shown in Figure 5, there are three operations on a ghost counter: `g_new_ctr`, `g_inc`, and `g_dec`, where `g_new_ctr` allocates a new ghost counter whose *value* is zero and returns a *ghost counter identifier* g for it; `g_inc(g)` increments the value of the ghost counter with identifier g and produces a *ticket* for the counter; and `g_dec(g)` consumes a ticket for the ghost counter g and decrements

⁶ Some logics for program verification, such as Iris [16], 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* [16, p. 5].

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 g is always equal to the number of tickets for g in the system. Similarly, to meet the second condition we use a *ghost monotonic counter*, which is similar to a ghost counter but does not have an operation `g_dec`.

As shown in Figures 6 and 7 the precondition and the postcondition of each routine are denoted by **req** and **ens**. Note that we use separation logic [26] to reason about the ownership of permissions. As shown in the routine `new_tvm`, after creating the lock l a permission `ulock` with two bags Wt and Ot , and a set It , which are initially empty, is produced for this lock. Additionally, we assign an arbitrary level, which is a level lower than the level of v , to this lock. Using the ghost commands `g_new_ctr` and `g_new_ictr` a ghost counter gt and a ghost monotonic counter gn is added to the resources of the current thread.

After creating and initializing the condition variable v a permission `cond` is produced for this condition variable where no permission but an obligation for v is transferred through any notification on this condition variable. We also assign r and l as the level and the associated lock of this condition variable, respectively. After creating the resulting `tvm` structure t , we initialize the lock l by consuming the assertion resulting from applying the invariant of l to the bags and the set in the permission `ulock`, i.e. $\text{linv}(t)(\{\}, \{\}, \{\})$. Note that gt and gn are two ghost fields added to the structure `tvm` that hold the identifiers of the ghost counters gt and gn . Lastly, we encapsulate all the resulting permissions in a predicate `tvm`. This predicate is used in the precondition of the routine `enter`. Another precondition of this routine is that the level v must be lower than the levels of all obligations of the current thread ($v \prec O$), because this thread might wait for v and l (note that the level of l is lower than the level of v , which by $v \prec O$ implies $l \prec O$). After acquiring l , this lock is loaded onto the obligations of the current thread and the invariant of this lock is produced. After incrementing the ghost counters gt and gn (we leak a ticket $\text{itic}(gn)$), and (conditionally) loading an obligation of v , the invariant of the loop, denoted by inv , holds. Note that this invariant ensures that if the body of the loop is executed ($t.\text{active} < \text{mine}$) there exists an obligation for v in the system ($0 < Ot(v)$), which is implied by $\text{mine} < \text{next}$. After execution of `wait(v, l)` there exists two bags Wt' and Ot' and a set It' such that the permissions described by $\text{linv}(t)(Wt', Ot', It')$ is produced for this thread. If $(v, \text{mine}) \notin It'$, which means $\text{mine} \leq \text{active}$, then this thread receives the transferred obligations, which is $\{v\}$. Otherwise, if $(v, \text{mine}) \in It'$, which means $\text{active} < \text{mine}$, it does not receive these obligations and again waits for v . Lastly, after releasing l the post condition of this routine shows that an obligation of v is loaded onto the current thread and a permission $\text{tic}(gt)$ is produced. As shown in the specification of the routine `leave`, this routine requires a permission $\text{tic}(gt)$ and discharges an obligation for v . This permission ensures that if a thread is waiting to enter ($\text{active} + 1 < \text{next}$) the set It is not empty ($0 < |It|$), which makes it possible to transfer an obligation of v through the command `notifyAll`. If there is no thread waiting to enter we can safely discharge an obligation for v . Before releasing the lock l , we decrement the counter gt to

$\text{tvm}(\text{tvm } t, \text{gctr } gt, \text{cond } v) ::= \text{lock}(t.l) * \text{cond}(v, \text{true}, \{v\}) \wedge$
 $R(t.l) < R(v) \wedge L(v) = t.l \wedge I(t.l) = \text{linv}(t) \wedge gt = t.gt \wedge v = t.v$

$\text{linv}'(\text{tvm } t) ::= \lambda Wt. \lambda Ot. \lambda It.$
 $\exists active, next. t.active \xrightarrow{5} active * m.next \xrightarrow{5} next \wedge active \leq next * \text{ictr}(t.gn, next) \wedge$
 $(active < next \Rightarrow 0 < Ot(t.v)) \wedge$
 $(next \leq active + 1 \Rightarrow Wt(t.v) = 0) \wedge$
 $It = \{(v, id) \mid active < id < next\} *$
 $\exists Tr. \text{ctr}(t.gt, Tr) \wedge (active + 1 < next \Rightarrow next - active - 1 \leq Tr \wedge Tr \leq |It|)$

$\text{linv}(\text{tvm } t) ::= \lambda Wt. \lambda Ot. \lambda It.$
 $\text{linv}'(t)(Wt, Ot, It) * \exists ac, nx. t.active \xrightarrow{5} ac * m.next \xrightarrow{5} nx$

routine `new_tvm()`{
req : {true}
 $l := \text{new_lock}();$
 $\{\text{ulock}(l, \emptyset, \emptyset, \emptyset) \wedge R(l) = r - 1\} \text{ } gt := \text{g_new_ctr}; \text{ } gn := \text{g_new_ictr};$
 $\{\text{ulock}(l, \emptyset, \emptyset, \emptyset) * \text{ctr}(gt, 0) * \text{ictr}(gn, 0)\}$
 $v := \text{new_cvar}(); \text{g_initc}(v);$
 $\{\text{ulock}(l, \emptyset, \emptyset, \emptyset) * \text{ctr}(gt, 0) * \text{ictr}(gn, 0) * \text{cond}(v, \text{true}, \{v\}) \wedge R(v) = r \wedge L(v) = l\}$
 $t := \text{tvm}(l := l, v := v, active := \text{new_int}(0), next := \text{new_int}(0), gt := gt, gn := gn);$
 $\text{g_initl}(l); t$
ens : $\{\lambda t. \text{tvm}(t, gt, v) \wedge R(v) = r\}$

routine `enter(tvm t)`{
req : $\{\text{obs}(O) * \text{tvm}(t, gt, v) \wedge v \prec O\}$
 $\text{acquire}(t.l);$
 $\{\text{obs}(O \uplus \{l\}) * \text{tvm}(t, gt, v) * \exists Wt, Ot, It. \text{locked}(l, Wt, Ot, It) * \text{linv}(t)(Wt, Ot, It)\}$
 $mine := t.next ++; \text{g_ictr_inc}(t.gn); \text{g_ctr_inc}(gt);$
 $\text{if}(t.next = t.active + 1) \text{g_charge}(v);$
inv : $\text{tvm}(t, gt, v) * \text{tic}(gt) * \exists active, next. t.active \xrightarrow{5} active * m.next \xrightarrow{5} next \wedge$
 $mine < next * \exists Wt, Ot, It. \text{locked}(l, Wt, Ot, It) *$
 $mine \leq active ? \text{obs}(O \uplus \{l, v\}) * \text{linv}'(t)(Wt, Ot, It) :$
 $\text{obs}(O \uplus \{l\}) * \text{linv}'(t)(Wt \uplus \{v\}, Ot, It \cup \{(v, mine)\})$
 $\text{while}(t.active < mine)$
 $\text{wait}(t.v, t.l);$
 $\{\text{obs}(O \uplus \{l, v\}) * \text{tvm}(t, gt, v) *$
 $\exists Wt, Ot, It. \text{locked}(l, Wt, Ot, It) * \text{linv}(t)(Wt, Ot, It) * \text{tic}(gt)\}$
 $\text{release}(t.l)$
ens : $\{\text{obs}(O \uplus \{v\}) * \text{tvm}(t, gt, v) * \text{tic}(gt)\}$

Fig. 6. The proof of deadlock-freedom of the program shown in Figure 1 (part one).

be able to consume the assertion resulting from applying the invariant of l to the bags and the set in the related **locked** permission. The routine **main** is deadlock-free, because starting from an empty bag of obligations, the verification of this routine ends with such a bag too.

```

routine leave(tvm  $t$ ){
  req : { $\text{obs}(O \uplus \{v\}) * \text{tvm}(t, gt, v) * \text{tic}(gt) \wedge v \prec O$ }
  acquire( $t.l$ );
  { $\text{obs}(O \uplus \{l, v\}) * \text{tic}(gt) * \text{tvm}(t, gt, v) * \exists Wt, Ot, It. \text{locked}(l, Wt, Ot, It) * \text{linv}(t)(Wt, Ot, It)$ }
   $t.active++$ ; if( $t.next = t.active$ )  $g\_discharge(v)$ ;
  notifyAll( $t.v$ );  $g\_ctr\_dec(gt)$ ;
  { $\text{obs}(O \uplus \{\}$ ) *  $\text{tvm}(t, gt, v) * \exists Wt, Ot, It. \text{locked}(l, Wt, Ot, It) * \text{linv}(t)(Wt, Ot, It)$ }
  release( $t.l$ )
  ens : { $\text{obs}(O) * \text{tvm}(t, gt, v)$ }

routine main(){
  req : { $\text{obs}(\{\})$ }
   $t := \text{new\_tvm}$ ;
  { $\text{obs}(\{\}) * \text{tvm}(t, gt, v) \wedge R(v) = r$ }
  fork(
    inv : { $\text{obs}(\{\}) * \text{tvm}(t, gt, v)$ }
    while(1){
      enter( $t$ ); // get service
      { $\text{obs}(\{v\}) * \text{tvm}(t, gt, v) * \text{tic}(gt)$ }
      leave( $t$ ));
    { $\text{obs}(\{\}) * \text{tvm}(t, gt, v)$ }
    enter( $t$ ); // get service
    { $\text{obs}(\{v\}) * \text{tvm}(t, gt, v) * \text{tic}(gt)$ }
    leave( $t$ )
    ens : { $\text{obs}(\{\})$ }
  }

```

Fig. 7. The proof of deadlock-freedom of the program shown in Figure 1 (part two).

4 Publishable Resources

In Section 3 we proved that the program in Figure 1 is deadlock-free, i.e. there is at least one thread in this program running. However, a stronger liveness property of this program is starvation-freedom, which implies that any thread calling **enter** eventually gets service (under the fairness assumption that every thread is scheduled to execute infinitely often). To achieve this goal we need to make sure that the loop in the routine **enter** eventually terminates, which means we need to provide a *variant* (ranking function) for this loop, e.g. an expression whose range is restricted to the non-negative integers and its value is decreased in each iteration of the loop.

$$\begin{array}{l}
\text{NEWPCOUNTER} \\
\{\text{true}\} \text{ g_new_pctr } \{\lambda g. \text{ pctr}(g, 0)\} \\
\\
\text{INCPOUNTER} \\
\{\text{pctr}(g, n)\} \text{ g_pinc}(g) \{\lambda _. \text{ pctr}(g, n+1) * \text{ptic}(g, 1, 0)\} \\
\\
\text{DECPOUNTER} \\
\{\text{pctr}(g, n) * \text{ptic}(g, n', p) \wedge 0 < n'\} \text{ g_pdec}(g) \{\lambda _. \text{ctr}(g, n-1) * \text{ptic}(g, n'-1, p) \wedge n' \leq n\} \\
\\
\text{MINIMUM} \\
\text{pctr}(g, n) * \text{ptic}(g, n', p) \Rightarrow \text{pctr}(g, n) * \text{ptic}(g, n', p) \wedge n' + p \leq n \\
\\
\text{MERGE} \\
\text{ptic}(g, n_1, p_1) * \text{ptic}(g, n_2, p_2) \Leftrightarrow \text{ptic}(g, n_1+n_2, p_1+p_2) \\
\\
\text{publishable}(\text{ptic}(g, n, p)) = \text{ptic}(g, 0, n) \quad \text{unpublishable}(\text{ptic}(g, n, p)) = \text{ptic}(g, 0, p+n)
\end{array}$$

Fig. 8. Publishable ghost counters.

Such a variant can be established for this loop if after each broadcast any thread waiting for v is ensured that the new value of the variable *active* is greater than the one before executing the wait command. This *information* must be transferred from the notifying thread to *all* of the waiting threads.

A first attempt to transfer this information is to create a ghost counter gp , keeping track of the value of *active*, and to transfer a ticket for this counter from the notifying threads to the ones notified. However, this attempt fails because the notifying thread requires $Wt(v)$ instances of this ticket (while it can produce only one instance of this ticket).

To address this problem, instead of a ghost counter, we use a *publishable ghost counter*⁷, shown in Figure 8. Tickets of this counter consist of two values; the first value indicates the number of the tickets of the current thread which have not been published to any other thread, and the second value indicates the number of the tickets which have been already published to some other threads. A permission $\text{ptic}(g, u, p)$, for example, indicates that the current thread has a total of $u+p$ tickets for g ; u tickets unpublished and p tickets published. Accordingly, the value of the counter, shown in the predicate pctr indicates the maximum number of the tickets that a thread can have. Obviously, a thread having a ticket $\text{ptic}(g, u, p)$ can publish u tickets for g ($\text{ptic}(g, 0, u)$), and after publishing these tickets it has a ticket $\text{ptic}(g, 0, p+u)$, which is not publishable anymore. Accordingly, we update the proof rule for **notifyAll**, as shown in Figure 9, such that it transfers a publishable part of a resource from the notifying thread to the notified ones, and makes this resource unpublishable.

Having this rule, it is possible that the notifying thread in Figure 1 having a ticket $\text{ptic}(gp, 1, 0)$, publishes a ticket $\text{ptic}(gp, 0, 1)$ to all of the waiting threads

⁷ This counter is an instance of a *max and plus united monoid* [22].

and informs them that the value of *active* has been increased, where *gp* is a publishable ghost counter keeping track of the value of *active*. Accordingly, the verification of this program is shown in Figure 10 where the loop in the routine *enter* has an appropriate variant implying the termination of that loop⁸. As shown in the routine *new_tvm* in this figure, a publishable ghost counter *gp* is created which keeps track of the value of *active*. As shown in the routine *enter*, this counter as well as a ticket *ptic(gp, 0, activemin)* is asserted in the invariant of the loop. Note that a ticket *ptic(gp, 0, 0)* can be produced by increasing and then decreasing the counter *gp*. Since whenever the thread is notified it receives a ticket *ptic(gp, 0, 1)* from the notifier thread, the value of *activemin* in this ticket is increased (using the Rule MERGE in Figure 8), which implies that the variant of the loop (*mine - active*) is decreased in each iteration. Additionally, by Rule MINIMUM we have *activemin* ≤ *active*, and by the loop invariant we have *active* ≤ *mine*, which implies that after each iteration the variant of the loop remains non-negative (*activemin* ≤ *mine*). As shown in the routine *leave*, since the variable *active* is increased, we can obtain a ticket *ptic(gp, 1, 0)* using the ghost command *g_pctr_inc(t, gp)*. The publishable part of this ticket (*ptic(gp, 0, 1)*) is transferred to the waiting threads through the command *notifyAll*, and the unpublishable part (*ptic(gp, 0, 1)*) is leaked in this routine.

NOTIFYALL
 $\{ \text{obs}(O \uplus ((v, id) \in It ? M' : \{\})) * \text{cond}(v, M, M') * \text{locked}(L(v), Wt, Ot, It) * M_p \wedge$
 $\quad \text{publishable}(M_p) = M \} \text{notifyAll}(v)$
 $\{ \lambda _. \text{obs}(O) * \text{cond}(v, M, M') * \text{locked}(L(v), Wt[v:=0], Ot, It - \{(v, id)\}) *$
 $\quad \text{unpublishable}(M_p) \}$

Fig. 9. Transferring publishable resources through broadcasts, where *publishable*(*M_p*) denotes the publishable part of the resource *M_p* and *unpublishable*(*M_p*) denotes the unpublishable part of *M_p* which remains after publishing *M_p*.

Publishable points-to resource. The notion of publishable resource can be also used to prove some other properties. For example, consider a publishable version of a points-to permission $x \xrightarrow{(u,p)} v$, where *u* is the fraction of the ownership of the location *x* which has not been published, *p* is the fraction of the ownership of the location *x* which has been published, $\text{publishable}(x \xrightarrow{(u,p)} v) = x \xrightarrow{(0,u)} v$, and $\text{unpublishable}(x \xrightarrow{(u,p)} v) = x \xrightarrow{(0,p+u)} v$. Although a resource $x \xrightarrow{(0,1)} v$ gives no information about the value stored in the location *x* and gives no permission to access *x*, a resource $x_1 \xrightarrow{(0,1)} v_1 * x_2 \xrightarrow{(0,1)} v_2$ ensures that $x_1 \neq x_2$. However, transferring or publishing these publishable and unpublishable resources through other synchronizations such as locks and channels requires future investigations.

⁸ Note that this loop terminates because 1) (we assume that) each thread is scheduled to execute infinitely often, 2) for any thread waiting for *v* there is an obligation for *v* which is discharged in a finite number of steps, and 3) each time that an obligations for *v* is discharged (and *v* is notified) the variant of this loop is decreased.

```

tvm(tvm t, gctr gt, cond v) ::= lock(t.l) * cond(v, ptic(t.gp, 0, 1), {v}) ∧
R(t.l) < R(v) ∧ L(v)=t.l ∧ l(t.l) = linv(t) ∧ gt = t.gt ∧ v = t.v

linv'(tvm t) ::= λWt. λOt. λIt.
  ∃active, next. t.active  $\xrightarrow{5}$  active * m.next  $\xrightarrow{5}$  next ∧ active ≤ next * ictr(t.gn, next)
  ∧ (active < next ⇒ 0 < Ot(t.v)) ∧ (next ≤ active + 1 ⇒ Wt(t.v) = 0) ∧
  It = {(v, id) | active < id < next} *
  ∃Tr. ctr(t.gt, Tr) ∧ (active + 1 < next ⇒ next - active - 1 ≤ Tr ∧ Tr ≤ |It|)

linv(tvm t) ::= λWt. λOt. λIt. linv'(t)(Wt, Ot, It) *
  ∃active, next. t.active  $\xrightarrow{5}$  active * m.next  $\xrightarrow{5}$  next * pctr(t.gp, active)

routine new_tvm() { req : {true}
  l := new_lock();
  {ulock(l, {}, {}, { }) ∧ R(l)=r - 1} gt := g_new_ctr; gn := g_new_ictr; gp := g_new_pctr;
  {ulock(l, {}, {}, { }) * ctr(gt, 0) * ictr(gn, 0) * pctr(gp, 0)}
  v := new_cvar(); g_initc(v);
  {ulock(l, {}, {}, { }) * ctr(gt, 0) * ictr(gn, 0) * pctr(gp, 0) *
   cond(v, ptic(gp, 0, 1), {v}) ∧ R(v)=r ∧ L(v)=l}
  t:=tvm(l:=l, v:=v, active := new_int(0), next := new_int(0), gt:=gt, gn:=gn, gp:=gp);
  g_initl(l); t ens : {λt. tvn(t, gt, v) ∧ R(v)=r}

routine enter(tvm t) { req : {obs(O) * tvn(t, gt, v) ∧ v < O}
  acquire(t.l);
  {obs(O ⊔ {l}) * tvn(t, gt, v) * ∃Wt, Ot, It. locked(l, Wt, Ot, It) * linv(t)(Wt, Ot, It)}
  mine := t.next++; g_ictr_inc(t.gn); g_ctr_inc(gt);
  if(t.next = t.active+1) g_charge(v);
  inv : tvn(t, gt, v) * tic(gt) * ∃active, next. t.active  $\xrightarrow{5}$  active * m.next  $\xrightarrow{5}$  next *
    ∃activemin. pctr(t.gp, active) * ptic(t.gp, 0, activemin) ∧
    mine < next * ∃Wt, Ot, It. locked(l, Wt, Ot, It) *
    mine ≤ active ? obs(O ⊔ {l, v}) * linv'(t)(Wt, Ot, It) :
    obs(O ⊔ {l}) * linv'(t)(Wt ⊔ {v}, Ot, It ∪ {(v, mine)})
  var : mine - activemin
  while(t.active < mine)
    wait(t.v, t.l);
  {obs(O ⊔ {l, v}) * tvn(t, gt, v) *
   ∃Wt, Ot, It. locked(l, Wt, Ot, It) * linv(t)(Wt, Ot, It) * tic(gt)}
  release(t.l)
  ens : {obs(O ⊔ {v}) * tvn(t, gt, v) * tic(gt)}

routine leave(tvm t) { req : {obs(O ⊔ {v}) * tvn(t, gt, v) * tic(gt) ∧ v < O}
  acquire(t.l);
  {obs(O ⊔ {l, v}) * tic(gt) * tvn(t, gt, v) *
   ∃Wt, Ot, It. locked(l, Wt, Ot, It) * linv(t)(Wt, Ot, It)}
  t.active++; g_pctr_inc(t.gp); if(t.next = t.active) g_discharge(v);
  notifyAll(t.v); g_ctr_dec(gt);
  {obs(O ⊔ {l}) * tvn(t, gt, v) * ∃Wt, Ot, It. locked(l, Wt, Ot, It) * linv(t)(Wt, Ot, It)}
  release(t.l)
  ens : {obs(O) * tvn(t, gt, v)}

```

Fig. 10. The proof of termination of the monitor shown in Figure 1.

5 Related Works

Separation Logic. Jung *et al.* [16] proposed a concurrent separation logic, namely *Iris*, for reasoning about safety of concurrent programs, as the logic in logical relations, to reason about type-systems and data-abstraction. In this logic user-defined protocols on shared state are expressed through *partial commutative monoids* and is enforced through *invariants*. However, the *Iris* program logic and many other logics such as [25, 23] only prove per-thread safety (i.e. no thread ever crashes): their adequacy theorems state that the program does not reach a state where some thread cannot make a step. This works because these logics do not consider blocking constructs, where a thread may legitimately be stuck temporarily.

Liveness properties. Several approaches to verify termination [21, 5, 28], total correctness [3], and lock freedom [10] 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 monitors and channels, where the suspension of a notifying/sending thread might lead a waiting thread to be infinitely blocked. In [24] 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 monitors because a waiting thread that is never notified cannot be considered as a terminating thread.

Verification of monitors through model checking. There are also some other approaches addressing some common synchronization bugs of programs in the presence of condition variables. In [35], 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 these commands into account. In other words, it might accept an undesired execution trace where the waiting thread is scheduled after the notifying thread, that might lead the waiting thread to be infinitely suspended. [17] 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, [1, 4] have 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 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.

Modular verification of channels. Inspired by the notion of capabilities [19, 18] and implicit dynamic frames [31, 29, 30], Leino *et al.* later integrated deadlock prevention into a verification system for an object-oriented and imperative programming language. 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 discharge an obligation for that channel. A thread can discharge an obligation for a channel if it either sends a message on that channel or delegates that obligation to another thread. However this approach is not general enough to support condition variables, since, unlike channels, a notification on a condition variable is lost if there is no thread waiting for that condition variable.

Modular verification of monitors. Recently, Hamin *et al.* [6, 7] introduced a modular approach to verify deadlock-freedom of monitors. Later, in [9] this approach is extended by allowing obligations to be transferred through notifications. However, in this approach obligations can be transferred using `notify(v)` but not using `notifyAll(v)`. This constraint does not allow this approach to verify monitors implementing queue locking algorithms, for example, where the releasing thread notifies all the waiting threads. Additionally, in this approach it is impossible to transfer a permission, indicating that the awoken threads are approaching the front of the waiting queue, from the notifying thread to all of the notified threads. In this paper we introduce a mechanism which allows obligations to be transferred from the notifying thread, executing `notifyAll`, to one of the notified threads. Additionally, we introduce a notion of publishable resources which can be published from a thread to many other threads.

6 Conclusion

In this paper we introduce two mechanisms to transfer obligations and publishable resources through broadcasts. The first mechanism allows to transfer an obligation through `notifyAll` from the thread holding a resource to the next thread, which acquires that resource. The second mechanism allows to publish a resource from a notifying thread to many other notified threads. These mechanisms allow the modular approaches, verifying deadlock-freedom of monitors, to verify a wider range of interesting programs. Additionally, these mechanisms allows these approaches to verify termination of the loop in which the wait command is executed (the loop that checks the waiting condition), which ensures that a thread will not infinitely wait in a monitor. The notion of publishable resources introduced in this paper, used to prove termination of the loops, can be also used to prove some other properties. However, transferring or publishing these resources through other synchronizations such as locks and channels requires future investigations.

References

1. 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. pp. 3–19. Springer (2016)
2. Craig, T.: Building fifo and priorityqueuing spin locks from atomic swap. Tech. rep., Technical Report TR 93-02-02, University of Washington, 02 1993.(ftp tr/1993 (1993)
3. D’Osualdo, E., Farzan, A., Gardner, P., Sutherland, J.: Tada live: Compositional reasoning for termination of fine-grained concurrent programs. arXiv preprint arXiv:1901.05750 (2019)
4. Gomes, P.d.C., Gurov, D., Huisman, M., Artho, C.: Specification and verification of synchronization with condition variables. *Science of Computer Programming* **163**, 174–189 (2018)
5. 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. pp. 110–117. IEEE (2016)
6. Hamin, J., Jacobs, B.: Deadlock-free monitors. In: European Symposium on Programming. pp. 415–441. Springer (2018)
7. Hamin, J., Jacobs, B.: Deadlock-free monitors: extended version. TR CW712, Department of Computer Science, KU Leuven, Belgium. Full version available at <http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW712.abs.html> (2018)
8. Hamin, J., Jacobs, B.: Deadlock-free monitors and channels (Jun 2019). <https://doi.org/10.5281/zenodo.3363702>, <https://doi.org/10.5281/zenodo.3363702>
9. Hamin, J., Jacobs, B.: Transferring Obligations Through Synchronizations. In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 19:1–19:58. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). <https://doi.org/10.4230/LIPIcs.ECOOP.2019.19>, <http://drops.dagstuhl.de/opus/volltexte/2019/10811>
10. 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. pp. 124–133. IEEE (2013)
11. Jacobs, B.: Provably live exception handling. In: Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs. p. 7. ACM (2015)
12. Jacobs, B.: Verifast 18.02. zenodo, <http://doi.org/10.5281/zenodo.1182724> (2018). <https://doi.org/10.5281/zenodo.1182724>, <http://doi.org/10.5281/zenodo.1182724>
13. Jacobs, B., Bosnacki, D., Kuiper, R.: Modular termination verification of single-threaded and multithreaded programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **40**(3), 12 (2018)
14. 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**, 41–55 (2011)
15. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the verifast program verifier. *Programming Languages and Systems* pp. 304–311 (2010)
16. 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), 637–650 (2015)

17. Kavi, K.M., Moshtaghi, A., Chen, D.J.: Modeling multithreaded applications using petri nets. *International Journal of Parallel Programming* **30**(5), 353–371 (2002)
18. Kobayashi, N.: A type system for lock-free processes. *Information and Computation* **177**(2), 122–159 (2002)
19. Kobayashi, N.: A new type system for deadlock-free processes. In: *CONCUR*. vol. 6, pp. 233–247. Springer (2006)
20. Leino, K.R.M., Müller, P., Smans, J.: Deadlock-free channels and locks. In: *European Symposium on Programming*. pp. 407–426. Springer (2010)
21. 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)*. p. 65. ACM (2014)
22. Mokhov, A.: United monoids (2019), <https://blogs.ncl.ac.uk/andreymokhov/united-monoids/>
23. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Communicating state transition systems for fine-grained concurrent resources. In: *European Symposium on Programming Languages and Systems*. pp. 290–310. Springer (2014)
24. Popeea, C., Rybalchenko, A.: Compositional termination proofs for multi-threaded programs. In: *TACAS*. vol. 12, pp. 237–251. Springer (2012)
25. Raad, A., Villard, J., Gardner, P.: Colosl: Concurrent local subjective logic. In: *European Symposium on Programming Languages and Systems*. pp. 710–735. Springer (2015)
26. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. pp. 55–74. IEEE (2002)
27. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P., Sutherland, J.: Modular termination verification for non-blocking concurrency. In: *ESOP*. pp. 176–201 (2016)
28. Rowe, R.N., Brotherston, J.: Automatic cyclic termination proofs for recursive procedures in separation logic. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. pp. 53–65. ACM (2017)
29. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. In: *Proceedings of the 10th ECOOP Workshop on Formal Techniques for Java-like Programs*. pp. 1–12 (2008)
30. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: *European Conference on Object-Oriented Programming*. pp. 148–172. Springer (2009)
31. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **34**(1), 2 (2012)
32. Specification, M.A.: Sleepconditionvariablecs function (2019), <https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-sleepconditionvariablecs>
33. Specification, O.A.: Using condition variables (2019), <https://docs.oracle.com/cd/E19120-01/open.solaris/816-5137/sync-21067/index.html>
34. Specification, T.S.U.A.: Wait on a condition (2019), http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_cond_wait.html
35. Wang, C., Hoang, K.: Precisely deciding control state reachability in concurrent traces with limited observability. In: *VMCAI*. pp. 376–394. Springer (2014)