Transferring Obligations Through Synchronizations

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

Bart Jacobs
imec-DistriNet, Department of Computer Science, KU Leuven, Belgium
bart.jacobs@cs.kuleuven.be

Abstract

One common approach for verifying safety properties of multithreaded programs is assigning appropriate permissions, such as ownership of a heap location, and obligations, such as an obligation to send a message on a channel, to each thread and making sure that each thread only performs the actions for which it has permissions and it also fulfills all of its obligations before it terminates. Although permissions can be transferred through synchronizations from a sender thread, where for example a message is sent or a condition variable is notified, to a receiver thread, where that message or that notification is received, in existing approaches obligations can only be transferred when a thread is forked. In this paper we introduce two mechanisms, one for channels and the other for condition variables, that allow obligations, along with permissions, to be transferred from the sender to the receiver, while ensuring that there is no state where the transferred obligations are lost, i.e. where they are discharged from the sender thread but not loaded onto the receiver thread yet. We show how these mechanisms can be used to modularly verify deadlock-freedom of a number of interesting programs, such as some variations of *client-server* programs, *fair readers-writers locks*, and *dining philosophers*, which cannot be modularly verified without such transfer. We also encoded the proposed separation logic-based proof rules in the VeriFast program verifier and succeeded in verifying the mentioned programs.

2012 ACM Subject Classification Theory of computation → Separation logic; Software and its engineering → Deadlocks; Software and its engineering → Process synchronization; Software and its engineering → Formal software verification; Theory of computation → Hoare logic

Keywords and phrases Hoare logic, separation logic, modular program verification, synchronization, transferring obligations, deadlock-freedom.

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.22

Funding This research received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453 (project VESSEDIA), as well as Flemish Research Fund project grant G.0962.17N.

Acknowledgements We thank three anonymous reviewers for their careful reading of our manuscript and their many insighful comments and suggestions, and also Amin Timany for his guidance on Coq.

1 Introduction

One common approach for verifying safety properties of multithreaded programs, such as absence of data races and deadlock, is assigning appropriate permissions [3, 37] and obligations [29, 33, 1] to each thread and making sure that each thread only performs the actions for which it has permissions and it also fulfills all of its obligations before it terminates. In a separation logic-based approach [42], for example, the ownership of a heap location is a permission for accessing that location, which is assigned to the thread allocating that location. Since there is only one instance of such permission, only one thread, the one having

such permission, can access that location, which ensures absence of data races. Absence of deadlock, as another example, can be verified by making sure that for any waitable object o, such as a lock, a channel, or a condition variable, for which a thread is waiting there is a thread obliged to fulfill an obligation for that object (which only waits for objects whose levels, some arbitrary numbers[1] associated with waitable objects, are lower than the level of o) [33, 2, 24, 15]. In this setting a thread can discharge an obligation for a lock, a channel, or a condition variable by releasing that lock, sending a message on that channel, or notifying that condition variable (under some conditions), respectively.

Permissions can be transferred through synchronizations [33, 34] by consuming them in the sender thread, where for example a message is sent or a condition variable is notified, and producing them in the receiver thread, where that message or that condition variable is received. However, this technique cannot be used for transferring obligations because the transferred obligations are lost if no thread receives them. Additionally, even in the presence of a receiver there might be a state where these obligations are discharged from the sender but not received by the receiver yet. If the obligations are transferred through channels, for example, if the receive is not scheduled to be executed *immediately* after the send then there would be a state where no thread holds the transferred obligations, which makes the approach unsound. Note that in the case that the send operation is synchronous [29], i.e. the sender thread is suspended until a thread receives the sent message, sending obligations through channels is perfectly fine because these obligation are never lost.

In this paper we introduce two mechanisms allowing threads to transfer their obligations through synchronizations while ensuring that there is no state where these obligations are lost, i.e. where they are discharged from the sender thread but not loaded onto the receiver thread yet. The main idea behind the first mechanism is that when the transferred obligations are discharged from the sender thread the levels of these obligations have been already loaded onto a (receiver) thread. These levels are discharged from the receiver thread when it receives the transferred obligations. In the second mechanism, which is specifically for condition variables, the obligations are discharged from a notifying thread only if there is a waiting thread which is notified and receives these obligations. We show that using these mechanisms in some modular approaches, verifying finite blocking [2] and deadlock-freedom of channels and locks [33, 24], semaphores [21], and monitors [15], enables them to verify a wider range of interesting programs, such as some variations of *client-server* programs, *fair mutexes*, *fair readers-writers locks*, and *dining philosophers*. We encoded the proposed proof rules in the VeriFast program verifier [25, 26, 22] and succeeded in verifying the programs above[2]. Additionally, we proved the soundness of both mechanisms (see Appendixes C and D): the soundness proof of the second mechanism is machine-checked with Coq[3].

In the rest of this paper Sections 2 and 3 introduce two mechanisms for transferring obligations through channels and notifications, Section 4 discusses related work, and Section 5 draws a conclusion.

[1] In this paper, for simplicity we use numbers as levels, but any partially ordered set can be used as the set of levels.
[2] The proof of these programs, verified by the VeriFast program verifier, can be found in [17].
[3] The soundness proof of the second mechanism, machine-checked in Coq, can be found in [17].

2 Transferring Obligations Through Channels

In this section we first review an approach, introduced by Leino *et al.* [33], which modularly verifies deadlock-freedom of channels. Then we extend this approach such that it also allows obligations to be transferred from the thread sending on a channel to the thread receiving from that channel. Lastly, we provide some variations of a client-server program which can be verified thanks to such extension.

2.1 Verifying Channels

Leino *et al.* [33] introduced a modular approach to verify deadlock-freedom of programs which communicate through channels. The main idea in this approach is to associate with each thread a bag[4] of channels, namely the bag of *obligations* of that thread, which must be empty when that thread terminates. A thread can discharge an obligation for a channel by either sending a message on that channel or delegating that obligation to a forked thread. This approach guarantees absence of deadlock by making sure that for any thread trying to receive a message from a channel ch there is either a message in ch or an obligation for ch in the bag of obligations of a thread which only waits for objects whose levels are lower than the level of ch (preventing circular dependencies). This constraint is applied by making sure that if a thread tries to receive from a channel ch then 1) it spends a *credit* for ch, where a thread can obtain a credit for ch by adding ch to the bag of its obligations, and 2) the level of ch is lower than the levels of the obligations of that thread. Note that a credit for a channel in this approach indicates that there is either a message on that channel or there is a thread holding an obligation for that channel.

As an example, consider the deadlock-free program shown in the left hand side of Figure 1, where after creating a channel ch, the main thread first forks a receiver thread, trying to receive a message from ch, and then sends a message on ch. The verification of this program is shown in the right hand side of this figure, where we use separation logic [42] to reason about the ownership of permissions. The verification of this program is started with an empty bag of obligations, denoted by $\mathsf{obs}(\{\!\})$. As indicated below each command, by creating a channel a (*duplicable* and *leakable*) permission $\mathsf{channel}$ for accessing that channel is produced and an arbitrary level, denoted by a function R, is assigned to that channel. Before forking the receiver thread, using a *ghost command*[5] $\mathsf{g\_credit}$, a credit and an obligation for this channel are loaded onto the main thread. The former is given to the forked thread, where this credit is spent by the command $\mathsf{receive}(ch)$, and the latter is discharged by the main thread when it executes the command $\mathsf{send}(ch, 12)$.

The separation logic-based proof rules, introduced by Jacobs *et al.* [23, 24], used to verify this program are shown in Figure 2. Note that the postcondition of each command in this figure is a lambda expression that given the return value of that command returns the assertion held after execution of that command. When a channel is created, as shown in Rule NewChannel, any arbitrary level can be assigned to that channel by the proof author. Note that, generally, this level must be chosen such that the constraint number 2, mentioned above, is met at each receive operation. Sending a message on a channel, as shown in Rule Send, discharges an obligation for that channel. As shown in Rule Receive, a thread can

[4] We model bags of objects as functions from objects to natural numbers. We also use $\uplus$ indicating the union (i.e. the pointwise sum) of two bags.

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

```
                                    {obs({})}
                                    ch := new_channel();
                                    {obs({}) * channel(ch) ∧ R(ch)=r}
                                    g_credit(ch);
                                    {obs({ch}) * channel(ch) * credit(ch)}
ch := new_channel();                fork(
fork(receive(ch));                    {obs({}) * channel(ch) * credit(ch) ∧ ch ≺ {}}
send(ch, 12)                          receive(ch)
                                      {obs({})}
                                    );
                                    {obs({ch}) * channel(ch)}
                                    send(ch, 12)
                                    {obs({})}
```

Figure 1 Verification of deadlock-freedom of channels, where O in $\mathsf{obs}(O)$ is the bag of obligations of the running thread, $\mathsf{R}(ch)$ denotes the level of ch, and $ch \prec O \Leftrightarrow \forall o{\in}O.\ \mathsf{R}(ch) < \mathsf{R}(o)$.

try to receive a message from a channel ch only if that thread spends a credit for ch and the level of ch is lower than the levels of all obligations of that thread. As shown in Rule Fork, a thread can transfer a part of its permissions and obligations to a forked thread, provided that the forked thread discharges all the delegated obligations. Lastly, as shown in Rule Credit, using a ghost command g_credit a thread can obtain a credit for a channel if that channel is loaded onto the bag of the obligations of that thread.

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 bag, never deadlocks, i.e. it always has a running thread, not waiting for any channel, until it terminates. We know that for any channel ch all of these proof rules preserve the following invariant, where $Ct(ch)$ and $Ot(ch)$ denote the total number of credits and obligations for ch in the system, respectively, and $\mathsf{size}(ch)$ denotes the number of messages in ch:

$$Ct(ch) \leqslant Ot(ch) + \mathsf{size}(ch) \tag{1}$$

Now consider a deadlocked state, where each thread of a verified program is waiting for a channel. Among all of these channels take the one having a minimal wait level, namely ch_{min}. Since $\mathsf{size}(ch_{min})$=0 and there exists a credit for ch_{min} in the system held by the waiting thread, according to the invariant above and the constraint number 2, there exists a thread having an obligation for ch_{min} that is waiting for a channel whose level is lower than the level of ch_{min}, which contradicts minimality of the level of ch_{min}.

2.2 Transferring Permissions and Obligations Through Channels

The approach presented in the previous section fails to verify some applications of channels where some obligations must be transferred from a thread sending on a channel to the thread receiving from that channel. Consider the *client-server* program shown in Figure 3, for example, where the server waits to receive a message that consists of a client request, which must be processed by the server, and a client channel, from which the client expects to receive the response of the server. Although the client routine in this example is deadlock-free, the Leino *et al.* approach fails to verify this program, since the routine main cannot give any credit for the channel ch', which is created inside the client, to the client. This program can be verified if after creating ch' a credit and an obligation for ch' are loaded onto the client

NewChannel
$\{\mathsf{true}\}\ \mathsf{new\_channel}\ \{\lambda ch.\ \mathsf{channel}(ch) \wedge \mathsf{R}(ch){=}r\}$

Send
$\{\mathsf{obs}(O) * \mathsf{channel}(ch)\}\ \mathsf{send}(ch, m)\ \{\lambda\_.\ \mathsf{obs}(O{-}\{\!|ch|\!\}) * \mathsf{channel}(ch)\}$

Receive
$\{\mathsf{obs}(O) * \mathsf{channel}(ch) * \mathsf{credit}(ch) \wedge ch \prec O\}\ \mathsf{receive}(ch)\ \{\lambda\_.\ \mathsf{obs}(O) * \mathsf{channel}(ch)\}$

Fork
$$\frac{\{a * \mathsf{obs}(O)\}\ c\ \{\lambda\_.\ \mathsf{obs}(\{\!|\}\!|)\}}{\{a * \mathsf{obs}(O \uplus O')\}\ \mathsf{fork}(c)\ \{\lambda\_.\ \mathsf{obs}(O')\}}$$

Credit
$\{\mathsf{obs}(O) * \mathsf{channel}(ch)\}\ \mathsf{g\_credit}(ch)\ \{\lambda\_.\ \mathsf{obs}(O \uplus \{\!|ch|\!\}) * \mathsf{channel}(ch) * \mathsf{credit}(ch)\}$

Figure 2 Proof rules ensuring deadlock-freedom of channels

```
routine server(channel ch){
 (req, ch') := receive(ch);
 result := process(req);
 send(ch', result)}
```

```
routine client(channel ch){
 ch' := new_channel();
 send(ch, (request(), ch'));
 receive(ch')}
```

```
routine main(){
 ch := new_channel();
 fork(server(ch));
 client(ch)}
```

Figure 3 A client-server program

such that the latter is transferred to the server through the client's request and the former is spent for the command $\mathsf{receive}(ch')$ executed in the client.

Similar to permissions, two necessary conditions for transferring obligations are: 1) when a thread sends a message on a channel ch the transferred obligations of ch, which are transferred through ch, are discharged from the bag of the obligations of that thread, and 2) when a thread receives a message from a channel ch the transferred obligations of ch are loaded onto the bag of the obligations of that thread. However, these conditions are not sufficient because these obligations are lost if no thread receives from ch. Additionally, even in the presence of a receiver if the $\mathsf{receive}(ch)$ is not scheduled to be executed *immediately* after the $\mathsf{send}(ch, m)$ then there would be a state where no thread holds the transferred obligations, which makes the approach unsound.

To address this problem, in addition to the bag of obligations, we associate with each thread a new bag, namely the bag of *importers* of that thread, which consists of the channels that transfer (import) some obligations to that thread. Similar to the bag of obligations, the bag of importers of a thread must be empty when that thread terminates and a thread can wait for a channel ch only if the level of ch is lower than the levels of all obligations which are *possibly* imported by all importers of that thread except for ch itself. Having this bag, we enforce an additional condition, numbered 3, when a thread sends some obligations on an importer channel, which imports some obligations, this channel must be already in the bag of importers of a (receiver) thread. This importer channel is discharged from the receiver thread as it receives a message from this channel. This additional condition is met by making sure that any thread sending on an importer channel ch spends a *transferring credit* for ch, where a thread can obtain a transferring credit of ch by adding ch to the bag

of its importers.

Formally, the third condition, which holds for any importer channel ch, ensures that for any transferring credit of ch or any message in ch (which means a transferring credit of ch has been spent by sending a message on ch and no thread has received it yet), there exists an instance of ch in the bag of importers of a (receiver) thread. This invariant is shown in the following as Invariant 2, where $\mathsf{M}^{\mathsf{r}}(ch)$ denotes the bag of the levels of the obligations which are *possibly* imported by ch, $Tt(ch)$ denotes the total number of transferring credits of ch in the system, $It(ch)$ denote the number of occurrences of ch in the bags of importers of all threads in the system, and $\mathsf{size}(ch)$ denotes the number of messages in the queue of ch.

$$\forall ch.\ \mathsf{M}^{\mathsf{r}}(ch) \neq \{\!\} \Rightarrow Tt(ch) + \mathsf{size}(ch) \leqslant It(ch) \tag{2}$$

Additionally, the two first conditions mentioned above, as well as the ones mentioned in the previous section, ensure that if a thread waits on a channel ch' then there is either an obligation of ch' in the system, or a message on ch', or a message in the queue of a channel through which an obligation of ch' is transferred (which means an obligation of ch' has been transferred through this message and no thread has received it yet). This invariant is formally shown in the following as Invariant 3, where $Ot(ch)$ and $Ct(ch)$ denote the total number of obligations and credits of ch in the system, $\mathsf{queue}(ch)$ denotes the list of messages in the channel ch, and $\mathsf{M}'(ch)$ is a function, that given a message, specifies the bag of the obligations which are transferred through that message in ch. Note that the levels of these obligations must be in the bag of the levels of the obligations which are possibly imported by ch, as shown formally in Invariant 4, where $\mathsf{levels}(O)$ maps each element of O to its level.

$$\forall ch'.\ Ct(ch') \leqslant Ot(ch') + \mathsf{size}(ch') + \sum_{ch} \sum_{m \in \mathsf{queue}(ch)} \mathsf{M}'(ch)(m)(ch') \tag{3}$$

$$\forall ch, m \in \mathsf{queue}(ch).\ \mathsf{levels}(\mathsf{M}'(ch)(m)) \subseteq \mathsf{M}^{\mathsf{r}}(ch) \tag{4}$$

It can be proved that any program preserving such invariants never deadlocks, i.e. it always has a running thread, not waiting for any channel, until it terminates. Consider a deadlocked state, where each thread of a program is waiting for a channel. Among all of these channels take the one having a minimal wait level, namely ch_{min}. Since $\mathsf{size}(ch_{min})=0$ and there exists a credit for ch_{min} in the system held by the waiting thread, according to Invariant 3, there exists either 1) a thread having an obligation of ch_{min} that is waiting for a channel whose level is lower than the level of ch_{min}, or 2) there exists a message m on a channel ch through which an obligation of ch_{min} is transferred, i.e. $0<\mathsf{M}'(ch)(m)(ch_{min})$ which by Invariant 4 implies $\mathsf{R}(ch_{min}) \in \mathsf{M}^{\mathsf{r}}(ch)$, where $\mathsf{R}(ch_{min})$ denotes the level of ch_{min}. In the first case minimality of the level of ch_{min} is contradicted. In the second case, since $0<\mathsf{size}(ch)$ and $\mathsf{M}^{\mathsf{r}}(ch) \neq \{\!\}$, by Invariant 2, there exists a thread having an importer channel ch that is waiting for a channel whose level is lower than the level of ch_{min} (because ch imports an obligation of level of ch_{min}), which again contradicts minimality of ch_{min}.

The proof rules enforcing such invariants are shown in Figure 4, where $\mathsf{M}^{\mathsf{r}}(ch)$ denotes the bag of the levels of the obligations which are possibly imported by ch, and the parameters M and M' in the permission $\mathsf{channel}$ of a channel are two functions that given a message return the permissions and the obligations which are transferred through that message. When a channel ch is created, as shown in Rule NewChannel, the value of these functions for this channel can be specified arbitrarily. As shown in Rule Send, when a message m is sent on ch, the permissions which are transferred through m as well as one transferring credit of

NewChannel
$\{\mathsf{true}\}\ \mathsf{new\_channel}\ \{\lambda ch.\ \mathsf{channel}(ch, M, M') \wedge \mathsf{R}(ch){=}r \wedge \mathsf{M}^{\mathsf{r}}(ch){=}R\}$

Send
$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch, M, M') * M(m) * (\mathsf{M}^{\mathsf{r}}(ch){=}\{\!\} \vee \mathsf{tranditt}(ch)) \wedge \mathsf{levels}(M'(m)) \subseteq \mathsf{M}^{\mathsf{r}}(ch)\}\ \mathsf{send}(ch, m)\ \{\lambda\_.\ \mathsf{obs}(O{-}\{\!|ch|\!\}{-}M'(m), I) * \mathsf{channel}(ch, M, M')\}$

Receive
$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch, M, M') * \mathsf{credit}(ch) \wedge ch \prec O \wedge ch \prec^{r} I\}\ \mathsf{receive}(ch)\ \{\lambda m.\ \mathsf{obs}(O{\uplus}M'(m), I{-}\{\!|ch|\!\}) * \mathsf{channel}(ch, M, M') * M(m)\}$

Fork
$$\frac{\{a * \mathsf{obs}(O, I)\}\ c\ \{\lambda\_.\ \mathsf{obs}(\{\!\}, \{\!\})\}}{\{a * \mathsf{obs}(O{\uplus}O', I{\uplus}I')\}\ \mathsf{fork}(c)\ \{\lambda\_.\ \mathsf{obs}(O', I')\}}$$

Credit
$\{\mathsf{obs}(O) * \mathsf{channel}(ch, M, M')\}\ \mathsf{g\_credit}(ch)\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{\!|ch|\!\}) * \mathsf{channel}(ch, M, M') * \mathsf{credit}(ch)\}$

Trandit
$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch, M, M')\}\ \mathsf{g\_tranditt}(ch)\ \{\lambda\_.\ \mathsf{obs}(O, I{\uplus}\{\!|ch|\!\}) * \mathsf{channel}(ch, M, M') * \mathsf{tranditt}(ch)\}$

Figure 4 The updated proof rules ensuring deadlock-freedom of importer channels, where $\mathsf{M}^{\mathsf{r}}(ch)$ denotes the bag of the levels of the obligations which are possibly imported by ch; M and M' in the permission channel of a channel are functions, that given a message, return the permissions and the obligations which are transferred through that message, respectively; bag I in the permission $\mathsf{obs}(O, I)$ of a thread denotes the channels importing some obligations onto that thread; $\mathsf{levels}(O)$ maps each element of O to its level; and $ch \prec^{r} I \Leftrightarrow \forall ch' {\in} I.\ ch{=}ch' \vee ch \prec \mathsf{M}^{\mathsf{r}}(ch')$.

ch, denoted by $\mathsf{tranditt}(ch)$, (if ch is an importer channel) are consumed and the obligations which are transferred through m as well as an obligation of ch are discharged from the bag of the obligations. Additionally, this rule makes sure by adding m to the queue of ch Invariant 4 is still preserved. As shown in Rule Receive, when a thread tries to receive a message m from ch the level of ch must be lower than the levels of the obligations of the thread and also the levels of the obligations which are possibly imported by all importers of that thread except for ch itself, i.e. $\forall ch' {\in} I.\ ch{=}ch' \vee ch \prec \mathsf{M}^{\mathsf{r}}(ch')$ where I is the bag of the importers of the receiving thread. Additionally, a credit of ch is consumed, the permissions which are transferred through m are produced, ch is discharged from the bag of the importers, and the obligations which are transferred through m are loaded onto the bag of the obligations. As shown in Rule Fork, a thread can transfer a part of its permissions, obligations, and importer channels to a forked thread, provided that the forked thread discharges all of the delegated obligations and importer channels. As shown in Rule Trandit a thread can obtain a transferring credit of ch by loading ch onto the bag of its importers.

The verification of the program in Figure 3 using the proposed proof rules is illustrated in Figure 5. Note that for the sake of readability we elide repeated occurrences of the permissions channel in the proof of the given programs. As shown in the precondition and the postcondition of the routine server, denoted by **req** and **ens**, this routine discharges an

importer channel ch, where a message from ch is received. Since this routine tries to wait on ch, it requires a credit and a permission channel for ch. The permission channel in the precondition of this routine indicates that along with the client's request the server receives an obligation and a permission channel for the client's channel through which no permission or obligation is transferred. The specification of the routine client indicates that this routine discharges an obligation for the server channel ch, since it sends a request to this channel. As it is shown in the verification of this routine, after creating the client channel ch' and before sending it to the server, one obligation and one credit for ch' are loaded onto the system. The former is transferred to the server, where it is discharged by sending a response on ch', and the latter is spent in the rest of this routine for receiving from ch'. The routine main in this program is successfully verified, since starting with an empty bag of obligations and importers the verification of this routine is finished with such bags too.

2.3 Conditional Channels

A different variation of a client-server program is shown in Figure 6, where a server keeps accepting the clients' requests until it receives a specific message done. The verification of this program using the proposed proof rules is illustrated in Figure 7. Note that if the client sends a message done to the server the session is finished and the client should not transfer an obligation of ch' through this message, i.e an obligation of ch' is transferred through a message sent on ch only if that message is not a message done. Since in this program a client can send multiple requests to the server and for each request it requires a permission trandit(ch), the server sends this permission to the client each time it replies to the client, i.e. a permission trandit(ch) is transferred through the client's channel ch'. Additionally, since the server might wait for ch more than once and for each wait it requires a credit(ch), before replying to the client a credit and an obligation for ch are loaded onto the server and the loaded obligation is transferred to the client, where this obligation is discharged by sending on ch, i.e. an obligation for ch is transferred through the client's channel ch'. Since the channel ch' transfers some obligations from the server to the client, the server requires a permission trandit(ch') before sending on ch', which can be obtained from the client through the channel ch, i.e. if the client sends a request (and not a message done) to the server it also sends a permission trandit(ch') to the server.

2.4 Server Channels

Another variation of a client-server program is shown in Figure 8, where a server *infinitely* accepts the clients' requests through a *server channel* s. A desired property of these kinds of programs is that if they have a thread waiting for a *non-server* channel they also have a running thread, not waiting for any channel. In other words, a program can be considered as a terminated program even if there are still some specific threads, namely *daemon threads*, such as a server thread or a garbage collector thread, which are not terminated. In Java these threads are terminated by the JVM when there is no longer any user thread running.

To achieve the mentioned property we only need to make sure that if a thread tries to receive from a server channel s it has no obligation and the bag of the importers of this thread only contains s.

It can be proved that any program verified by enforcing the constraint above meets the mentioned desired property, i.e. if this program has a thread waiting for a non-server channel it also has a running thread, not waiting for any channel. Consider an undesired state, where each thread is waiting for a channel while some of these channels are non-server channels.

```
Mch ::= λm. channel(snd(m), λ_. true, λ_. {|}) ∧ M^r(snd(m))={|}
M'ch ::= λm. {|snd(m)|}

routine server(channel ch){
req : {obs({|}, {|ch|}) * channel(ch, Mch, M'ch) * credit(ch)}
 (req, ch') := receive(ch);
 {obs({|ch'|}, {|}) * channel(ch', λ_. true, λ_. {|}) ∧ M^r(ch')={|}}
 result := process(req);
 send(ch', result)
ens : {obs({|}, {|})}}

routine client(channel ch){
req : {obs({|ch|}, {|}) * channel(ch, Mch, M'ch) * tranditch) ∧ M^r(ch)={|1|}}
 ch' := new_channel();
 {obs({|ch|}, {|}) * channel(ch', λ_. true, λ_. {|}) * tranditch) ∧ R(ch')=1 ∧ M^r(ch')={|}}
 g_credit(ch');
 {obs({|ch, ch'|}, {|}) * tranditch) * creditch')}
 send(ch, (request(), ch'));
 {obs({|}, {|}) * creditch')}
 receive(ch')
ens : {obs({|}, {|})}}

routine main(){
req : {obs({|}, {|})}
 ch := new_channel();
 {obs({|}, {|}) * channel(ch, Mch, M'ch) ∧ R(ch)=1 ∧ M^r(ch)={|1|}}
 g_credit(ch);
 {obs({|ch|}, {|}) * creditch)}
 g_tranditch);
 {obs({|ch|}, {|ch|}) * creditch) * tranditch)}
 fork(
   {obs({|}, {|ch|}) * creditch)}
   server(ch)
   {obs({|}, {|})});
 {obs({|ch|}, {|}) * tranditch)}
 client(ch)
ens : {obs({|}, {|})}}
```

Figure 5 Verification of the program in Figure 3

```
routine cserver(channel ch){
 (req, ch') := receive(ch);
 while(req ≠ done){
   send(ch', process(req));
   (req, ch') := receive(ch) }}

routine client(channel ch){
 ch' := new_channel();
 send(ch, (request(), ch'));
 receive(ch');
 send(ch, (done, ch'))}

routine main()
{
 ch := new_channel();
 fork(cserver(ch));
 client(ch)}
```

Figure 6 A server keeps serving a client until receiving a specific message done

$\mathsf{Mch}(ch) ::= \lambda m.\ \mathsf{channel}(\mathsf{snd}(m), \lambda\_.\ \mathsf{tranditt}(ch), \lambda\_.\ \{ch\}) \wedge \mathsf{M}^{\mathsf{r}}(\mathsf{snd}(m))=\{1\} *$
$\quad \mathsf{fst}(m)=\mathsf{done}\ ?\ \mathsf{true} : \mathsf{tranditt}(\mathsf{snd}(m))$
$\mathsf{M'ch} ::= \lambda m.\ \mathsf{fst}(m)=\mathsf{done}\ ?\ \{\} : \{\mathsf{snd}(m)\}$

routine cserver(channel ch){
req : $\{\mathsf{obs}(\{\}, \{ch\}) * \mathsf{channel}(ch, \mathsf{Mch}(ch), \mathsf{M'ch}) * \mathsf{credit}(ch) \wedge \mathsf{R}(ch)=1 \wedge \mathsf{M}^{\mathsf{r}}(ch)=\{1\}\}$
$(req, ch') := \mathsf{receive}(ch);$
$\{\mathsf{obs}(req=\mathsf{done}\ ?\ \{\} : \{ch'\}, \{\}) * \mathsf{channel}(ch', \lambda\_.\ \mathsf{tranditt}(ch), \lambda\_.\ \{ch\}) *$
$(req=\mathsf{done}\ ?\ \mathsf{true} : \mathsf{tranditt}(ch')) \wedge \mathsf{M}^{\mathsf{r}}(ch')=\{1\}\}$
$\mathsf{while}(req \neq \mathsf{done})\{$
inv : $\{\mathsf{M}^{\mathsf{r}}(ch')=\{1\} \wedge req=\mathsf{done}\ ?\ \mathsf{obs}(\{\}, \{\}) : (\mathsf{obs}(\{ch'\}, \{\}) * \mathsf{tranditt}(ch'))\}$
$\quad \mathsf{g\_tranditt}(ch);\ \mathsf{g\_credit}(ch);$
$\quad \{\mathsf{obs}(\{ch', ch\}, \{ch\}) * \mathsf{tranditt}(ch') * \mathsf{tranditt}(ch) * \mathsf{credit}(ch)\}$
$\quad \mathsf{send}(ch', \mathsf{process}(req));$
$\quad \{\mathsf{obs}(\{\}, \{ch\}) * \mathsf{credit}(ch)\}$
$\quad (req, ch') := \mathsf{receive}(ch)$
$\}$**ens** : $\{\mathsf{obs}(\{\}, \{\})\}\}$

routine client(channel ch){
req : $\{\mathsf{obs}(\{ch\}, \{\}) * \mathsf{channel}(ch, \mathsf{Mch}(ch), \mathsf{M'ch}) * \mathsf{tranditt}(ch) \wedge \mathsf{R}(ch)=1 \wedge \mathsf{M}^{\mathsf{r}}(ch)=\{1\}\}$
$ch' := \mathsf{new\_channel}();$
$\{\mathsf{obs}(\{ch\}, \{\}) * \mathsf{tranditt}(ch) * \mathsf{channel}(ch', \lambda\_.\ \mathsf{tranditt}(ch), \lambda\_.\ \{ch\}) \wedge \mathsf{R}(ch')=1 \wedge$
$\mathsf{M}^{\mathsf{r}}(ch')=\{1\}\}$
$\mathsf{g\_credit}(ch');\ \mathsf{g\_tranditt}(ch');$
$\{\mathsf{obs}(\{ch, ch'\}, \{ch'\}) * \mathsf{tranditt}(ch) * \mathsf{credit}(ch') * \mathsf{tranditt}(ch')\}$
$\mathsf{send}(ch, (\mathsf{request}(), ch'));$
$\{\mathsf{obs}(\{\}, \{ch'\}) * \mathsf{credit}(ch')\}$
$\mathsf{receive}(ch');$
$\{\mathsf{obs}(\{ch\}, \{\}) * \mathsf{tranditt}(ch)\}$
$\mathsf{send}(ch, (\mathsf{done}, ch'))$
ens : $\{\mathsf{obs}(\{\}, \{\})\}\}$

routine main(){
req : $\{\mathsf{obs}(\{\}, \{\})\}$
$ch := \mathsf{new\_channel}();$
$\{\mathsf{obs}(\{\}, \{\}) * \mathsf{channel}(ch, \mathsf{Mch}(ch), \mathsf{M'ch}) \wedge \mathsf{R}(ch)=1 \wedge \mathsf{M}^{\mathsf{r}}(ch)=\{1\}\}$
$\mathsf{g\_credit}(ch)\ ;\mathsf{g\_tranditt}(ch);$
$\{\mathsf{obs}(\{ch\}, \{ch\}) * \mathsf{credit}(ch) * \mathsf{tranditt}(ch)\}$
$\mathsf{fork}($
$\quad \{\mathsf{obs}(\{\}, \{ch\}) * \mathsf{credit}(ch)\}$
$\quad \mathsf{cserver}(ch)$
$\quad \{\mathsf{obs}(\{\}, \{\})\});$
$\{\mathsf{obs}(\{ch\}, \{\}) * \mathsf{tranditt}(ch)\}$
$\mathsf{client}(ch)$
ens : $\{\mathsf{obs}(\{\}, \{\})\}\}$

Figure 7 Verification of the program in Figure 6, where $a\ ?\ b : c$ evaluates to b if the value of a is true, and otherwise to c.

```
routine server(channel s){          routine client(channel s){          routine main(){
 while(true){                        ch' := new_channel();               s := new_channel();
   (req, ch') := receive(s);         send(s, (request(), ch'));          fork(server(s));
   send(ch', process(req))           receive(ch')                        fork(client(s));
 }}                                  }                                   client(s)}
```

Figure 8 A server keeps serving clients

NewChannel

$$\{\mathsf{true}\}\ \mathsf{new\_channel}\ \{\lambda ch.\ \mathsf{channel}(ch, M, M') \land \mathsf{R}(ch){=}r \land \mathsf{M^r}(ch){=}R \land \mathsf{S}(ch){=}b\}$$

Receive

$$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch, M, M') * (\mathsf{S}(ch) \lor \mathsf{credit}(ch)) \land ch \prec O \land ch \prec^r I \land (\neg\mathsf{S}(ch) \lor (O{=}\{\!\}\!\} \land \forall ch'{\in}I.\ ch'{=}ch))\}\ \mathsf{receive}(ch)$$
$$\{\lambda m.\ \mathsf{obs}(O{\uplus}M'(m), I{-}\{\!\{ch\}\!\}) * \mathsf{channel}(ch, M, M') * M(m)\}$$

Trandits

$$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch, M, M')\}\ \mathsf{g\_trandits}(ch)$$
$$\{\lambda\_.\ \mathsf{obs}(O, I{\uplus}\{\!\{ch^\infty\}\!\}) * \mathsf{channel}(ch, M, M') * \mathsf{trandit}^\infty(ch)\}$$

Figure 9 The updated proof rules ensuring deadlock-freedom of server channels, where S is a function that given a channel specifies whether that channel is a server channel or not, and o^∞ represents an infinite number of occurrences of o.

Among all of these non-server channels take the one having a minimal wait level, namely ch_{min}. Since $\mathsf{size}(ch_{min}){=}0$ and there exists a credit for ch_{min} in the system held by the waiting thread, by Invariant 3, either 1) there exists a thread having an obligation of ch_{min} that is waiting for a channel ch whose level is lower than the level of ch_{min}, or 2) there exists a message m in a channel ch through which an obligation of ch_{min} is transferred, i.e. $0{<}\mathsf{M'}(ch)(m)(ch_{min})$ which by Invariant 4 implies $\mathsf{R}(ch_{min}){\in}\mathsf{M^r}(ch)$, where $\mathsf{R}(ch_{min})$ denotes the level of ch_{min}. The first case contradicts minimality of the level of ch_{min}, because in this case ch is a non-server channel, since that thread is waiting for ch while it has some obligations. In the second case, since $0{<}\mathsf{size}(ch)$ and $\mathsf{M^r}(ch){\neq}\{\!\}\!\}$, by Invariant 2, there exists a thread t having an importer channel ch that is waiting for a channel ch_1 whose level is lower than the level of ch_{min} (because ch imports an obligation of level of ch_{min}). Since $0{<}\mathsf{size}(ch)$ we know $ch_1{\neq}ch$. Accordingly, ch_1 cannot be a server channel (because t is trying to receive from ch_1 while it has an importer channel ch which is not equal to ch_1), which is a contradiction.

The proof rules which need to be updated are shown in Figure 9. As shown in Rule NewChannel, when a channel is created it must be specified whether this channel is a server channel or not, which is denoted by a function S. As shown in Rule Receive, if a thread tries to receive from a server channel ch the bag of the obligations of this thread must be empty and the bag of the importer channels of this thread must only contain ch. Note that a thread does not need to spend a credit for waiting on a server channel. We also introduce a new ghost command $\mathsf{g\_trandits}(ch)$, shown in Rule Trandits, which produces an infinite number of transferring credits of ch, denote by $\mathsf{trandit}^\infty(ch)$, by loading an infinite number of ch, denoted by ch^∞, onto the importers. The verification of the program in Figure 8 using these rules is shown in Figure 10, where the server does not need any credit

$\mathsf{Ms}(s) ::= \lambda m.\ \mathsf{channel}(\mathsf{snd}(m), \lambda\_.\ \mathsf{tranditt}(s), \lambda\_.\ \{\!\}) \wedge \mathsf{M}^{\mathsf{r}}(\mathsf{snd}(m))=\{\!\}$
$\mathsf{M's} ::= \lambda m.\ \{\!|\mathsf{snd}(m)|\!\}$

routine server(channel s){
req : $\{\mathsf{obs}(\{\!\}, \{\!|s^\infty|\!\}) * \mathsf{channel}(s, \mathsf{Ms}(s), \mathsf{M's}) \wedge \mathsf{M}^{\mathsf{r}}(s)=\{\!|1|\!\} \wedge \mathsf{S}(s)\}$
while(true){
inv : $\{\mathsf{obs}(\{\!\}, \{\!|s^\infty|\!\})\}$
$(req, ch') := \mathsf{receive}(s)$;
$\{\mathsf{obs}(\{\!|ch'|\!\}, \{\!|s^\infty|\!\} - \{\!|s|\!\}) * \mathsf{channel}(ch', \lambda\_.\ \mathsf{tranditt}(s), \lambda\_.\ \{\!\}) \wedge \mathsf{M}^{\mathsf{r}}(ch')=\{\!\}\}$
g_tranditt(s);
$\{\mathsf{obs}(\{\!|ch'|\!\}, \{\!|s^\infty|\!\}) * \mathsf{tranditt}(s)\}$
send$(ch', \mathsf{process}(req))$
$\{\mathsf{obs}(\{\!\}, \{\!|s^\infty|\!\})\}$
}
ens : $\{\mathsf{false}\}\}$

routine client(channel s){
req : $\{\mathsf{obs}(\{\!\}, \{\!\}) * \mathsf{channel}(s, \mathsf{Ms}(s), \mathsf{M's}) * \mathsf{tranditt}(s) \wedge \mathsf{M}^{\mathsf{r}}(s)=\{\!|1|\!\}\}$
$ch' := \mathsf{new\_channel}()$;
$\{\mathsf{obs}(\{\!\}, \{\!\}) * \mathsf{tranditt}(s) * \mathsf{channel}(ch', \lambda\_.\ \mathsf{tranditt}(s), \lambda\_.\ \{\!\}) \wedge \mathsf{R}(ch')=1 \wedge \mathsf{M}^{\mathsf{r}}(ch')=\{\!\}\}$
g_credit(ch');
$\{\mathsf{obs}(\{\!|ch'|\!\}, \{\!\}) * \mathsf{tranditt}(s) * \mathsf{credit}(ch')\}$
send$(s, (\mathsf{request}(), ch'))$;
$\{\mathsf{obs}(\{\!\}, \{\!\}) * \mathsf{credit}(ch')\}$
receive(ch')
$\{\mathsf{obs}(\{\!\}, \{\!\}) * \mathsf{tranditt}(s)\}$
ens : $\{\mathsf{obs}(\{\!\}, \{\!\})\}\}$

routine main(){
req : $\{\mathsf{obs}(\{\!\}, \{\!\})\}$
$s := \mathsf{new\_channel}()$;
$\{\mathsf{obs}(\{\!\}, \{\!\}) * \mathsf{channel}(s, \mathsf{Ms}(s), \mathsf{M's}) \wedge \mathsf{M}^{\mathsf{r}}(s)=\{\!|1|\!\}\}$
g_tranditts(s);
$\{\mathsf{obs}(\{\!\}, \{\!|s^\infty|\!\}) * \mathsf{tranditt}^\infty(s)\}$
fork(
$\{\mathsf{obs}(\{\!\}, \{\!|s^\infty|\!\})\}$
server(s));
$\{\mathsf{obs}(\{\!\}, \{\!\}) * \mathsf{tranditt}^\infty(s)\}$
fork(client(s));
$\{\mathsf{obs}(\{\!\}, \{\!\}) * \mathsf{tranditt}^\infty(s)\}$
client(s)
ens : $\{\mathsf{obs}(\{\!\}, \{\!\})\}\}$

Figure 10 Verification of the program in Figure 8

```
routine server(channel s){
 while(true){
   (thr, ch') := receive(s);
   ch := new_channel();
   send(ch', (through, ch));
   fork(cserver(ch))
 }}
```

```
routine client(channel s){
 ch' := new_channel();
 send(s, (through, ch'));
 (thr, ch) := receive(ch');
 send(ch, (request(), ch'));
 (res, ch) := receive(ch');
 send(ch, (done, ch'))}
```

```
routine main()
{
 s := new_channel();
 fork(server(s));
 fork(client(s));
 client(s)
}
```

Figure 11 A server keeps serving clients through separate conditional channels

for waiting on the server channel s. Note that this verification ensures neither termination nor deadlock-freedom. It actually ensures that if this program has a thread waiting for the non-server channel ch' it also has a running thread, not waiting for any channel.

Using the proposed proof rules it is also possible to verify some other variations of client-server programs such as the one shown in Figure 11, where a server infinitely serves clients' request through separate conditional channels (see Appendix A illustrating the proof of this program). Note that the definition of the function cserver in this figure is similar to the one shown in Figure 6.

3 Transferring Obligations Through Notifications

In this section we introduce a mechanism which allows obligations to be transferred from a thread notifying a condition variable (CV) to the notified thread. The main idea behind this mechanism is based on this unique feature of condition variables that when a sender thread notifies a CV, if there is a receiver thread waiting for that CV, the receiver thread *immediately* receives this notification. Accordingly, if there exists a thread waiting for a condition variable v, it is safe to discharge the transferred obligations of v when v is notified and load these obligations to the receiver thread as it is notified. Note that a notification on a condition variable is lost if there is no thread waiting for that condition variable.

The number of threads waiting for a condition variable can be tracked by using the approach introduced by Hamin *et al.* [15, 16], which modularly verifies deadlock-freedom of programs synchronized by monitors. Since this approach only allows permissions to be transferred from a notifying thread to the one notified, it cannot verify some interesting programs such as a particular implementation of fair readers-writers locks, shown in Figure 19, and dining philosophers (see Appendix B). In the following we first review this approach and then we extend this approach such that it also allows transferring of obligations, enabling it to verify a wider range of programs such as the ones we just mentioned.

3.1 Verifying Monitors

Hamin *et al.* [15, 16] introduced a modular approach for verifying deadlock-freedom of programs synchronized by 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 a command notify(v)/notifyAll(v) wakes up one/all thread(s) waiting for CV v, if any. 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

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

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

Acquire
$\{\mathsf{lock}(l) * \mathsf{obs}(O) \wedge l \prec O\}\ \mathsf{acquire}(l)$
$\{\lambda\_.\ \exists Wt, Ot.\ \mathsf{locked}(l,\ Wt, Ot) * \mathsf{I}(l)(Wt, Ot) * \mathsf{obs}(O{\uplus}\{\!|l|\!\})\}$

Release
$\{\mathsf{locked}(l,\ Wt, Ot) * \mathsf{I}(l)(Wt, Ot) * \mathsf{obs}(O{\uplus}\{\!|l|\!\})\}\ \mathsf{release}(l)\ \{\lambda\_.\ \mathsf{lock}(l) * \mathsf{obs}(O)\}$

NewCV
$\{\mathsf{true}\}\ \mathsf{new\_cvar}\ \{\lambda v.\ \mathsf{ucond}(v) \wedge \mathsf{R}(v){=}r\}$

InitCV
$\{\mathsf{ucond}(v) * \mathsf{ulock}(l,\ Wt, Ot)\}\ \mathsf{g\_initc}(v)\ \{\lambda\_.\ \mathsf{cond}(v, M) * \mathsf{ulock}(l,\ Wt, Ot) \wedge \mathsf{L}(v){=}l\}$

Wait
$\{\mathsf{cond}(v, M) * \mathsf{locked}(l,\ Wt, Ot) * \mathsf{I}(l)(Wt{\uplus}\{\!|v|\!\}, Ot) * \mathsf{obs}(O{\uplus}\{\!|l|\!\}) \wedge$
$l{=}\mathsf{L}(v) \wedge v \prec O \wedge l \prec O \wedge \mathsf{enoughObs}(v,\ Wt{\uplus}\{\!|v|\!\}, Ot)\}\ \mathsf{wait}(v, l)$
$\{\lambda\_.\ \mathsf{cond}(v, M) * \mathsf{obs}(O{\uplus}\{\!|l|\!\}) * \exists Wt', Ot'.\ \mathsf{locked}(l,\ Wt', Ot') * \mathsf{I}(l)(Wt', Ot') * M\}$

Notify
$\{\mathsf{cond}(v, M) * \mathsf{locked}(\mathsf{L}(v),\ Wt, Ot) * (Wt(v){=}0 \vee M)\}\ \mathsf{notify}(v)$
$\{\lambda\_.\ \mathsf{cond}(v, M) * \mathsf{locked}(\mathsf{L}(v),\ Wt{-}\{\!|v|\!\}, Ot)\}$

NotifyAll
$\{\mathsf{cond}(v, M) * \mathsf{locked}(\mathsf{L}(v),\ Wt, Ot) * (\mathop{*}_{i:=1}^{Wt(v)} M)\}\ \mathsf{notifyAll}(v)$
$\{\lambda\_.\ \mathsf{cond}(v, M) * \mathsf{locked}(\mathsf{L}(v),\ Wt[v{:=}0], Ot)\}$

ChargeObligation
$\{\mathsf{obs}(O) * \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot)\}\ \mathsf{g\_chrg}(v)$
$\{\lambda\_.\ \mathsf{obs}(O{\uplus}\{\!|v|\!\}) * \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot{\uplus}\{\!|v|\!\})\}$

DischargeObligation
$\{\mathsf{obs}(O) * \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot) \wedge \mathsf{enoughObs}(v,\ Wt, Ot{-}\{\!|v|\!\})\}\ \mathsf{g\_disch}(v)$
$\{\lambda\_.\ \mathsf{obs}(O{-}\{\!|v|\!\}) * \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot{-}\{\!|v|\!\})\}$

Figure 12 Proof rules verifying deadlock-freedom of monitors, where $Wt(v)$ and $Ot(v)$ denote the total number of threads waiting for v and the total number of obligations for v, respectively; the parameter M in the permission cond of a condition variable denotes the permissions which are transferred from the thread notifying that condition variable to the one(s) notified; $\mathsf{L}(v)$ denotes the lock associated with the condition variable v; $\mathsf{enoughObs}(v, Wt, Ot) \Leftrightarrow (Wt(v){>}0 \Rightarrow Ot(v){>}0)$; and $v \prec O \Leftrightarrow \forall o{\in}O.\ \mathsf{R}(v) < \mathsf{R}(o)$.

the proof of that thread. In order to ensure the mentioned constraint this approach makes sure that 1) if a command $\mathsf{wait}(v, l)$ is executed then $0 < Ot(v)$, i.e. there is an obligation of v in the system, 2) if an obligation of 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 $\mathsf{wait}(v, l)$ 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 12. As shown in Rule NewLock, when a lock l is created an arbitrary level is assigned to that lock and an uninitialized lock permission $\mathsf{ulock}(l, \{\}, \{\})$ is produced. The second and the third parameters of this permission are two bags mapping the CVs associated with l to their number of waiting threads and obligations, respectively. As shown in Rule InitLock, this uninitialized lock permission can be converted to a (*duplicable and leakable*) lock permission $\mathsf{lock}(l)$ if the assertion resulting from applying the invariant of that lock, denoted by $\mathsf{I}(l)$, 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 $\mathsf{locked}(l, Wt, Ot)$ 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 and leakable*) permission cond if a lock is associated to this CV, denoted by $\mathsf{L}(v)$. Additionally, the transferred permissions of this CV, which are transferred from the notifying thread to the one notified, are also specified in this rule, denoted by M in the permission cond. These permissions are consumed when a command $\mathsf{notify}(v)$ is executed (if there is a thread waiting for v; see the precondition of Rule Notify), and are produced when a command $\mathsf{wait}(v, l)$ is executed (see the postcondition of Rule Wait). Note that $\mathsf{notifyAll}(v)$ transfers $Wt(v)$ instances of these permissions, denoted by $\overset{Wt(v)}{\underset{i:=1}{*}} M$ (see the precondition of Rule NotifyAll). As shown in Rule Wait, when a command $\mathsf{wait}(v, l)$ is executed, since l is going to be released and the number of 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, too, since when the thread is woken up it tries to reacquire l. As previously mentioned, the precondition of this rule also makes sure that $0 < Ot(v)$, which is enforced by the invariant $\mathsf{enoughObs}(v, Wt \uplus \{\!\{v\}\!\}, Ot)$. This invariant follows from $\mathsf{I}(l)(Wt \uplus \{\!\{v\}\!\}, Ot)$

InitCV
$$\{\mathsf{ucond}(v) * \mathsf{ulock}(l, Wt, Ot)\}\ \mathsf{g\_initc}(v)\ \{\lambda\_.\ \mathsf{cond}(v, M, M') * \mathsf{ulock}(l, Wt, Ot) \wedge \mathsf{L}(v){=}l\}$$

Wait
$$\{\mathsf{cond}(v, M, M') * \mathsf{locked}(l, Wt, Ot) * \mathsf{I}(l)(Wt{\uplus}\{\!|v|\!\}, Ot) * \mathsf{obs}(O{\uplus}\{\!|l|\!\}) \wedge l{=}\mathsf{L}(v) \wedge v{\prec}O \wedge l{\prec}O{\uplus}M' \wedge \mathsf{enoughObs}(v, Wt{\uplus}\{\!|v|\!\}, Ot)\}\ \mathsf{wait}(v, l)\ \{\lambda\_.\ \mathsf{cond}(v, M, M') * \mathsf{obs}(O{\uplus}\{\!|l|\!\}{\uplus}M') * \exists Wt', Ot'.\ \mathsf{locked}(l, Wt', Ot') * \mathsf{I}(l)(Wt', Ot') * M\}$$

Notify
$$\{\mathsf{obs}(O{\uplus}(0{<}Wt(v)\ ?\ M' : \{\!|\,|\!\})) * \mathsf{cond}(v, M, M') * \mathsf{locked}(\mathsf{L}(v), Wt, Ot) * (Wt(v){=}0 \vee M)\}\ \mathsf{notify}(v)\ \{\lambda\_.\ \mathsf{obs}(O) * \mathsf{cond}(v, M, M') * \mathsf{locked}(\mathsf{L}(v), Wt{-}\{\!|v|\!\}, Ot)\}$$

NotifyAll
$$\{\mathsf{cond}(v, M, \{\!|\,|\!\}) * \mathsf{locked}(\mathsf{L}(v), Wt, Ot) * (\mathop{*}_{i:=1}^{Wt(v)} M)\}\ \mathsf{notifyAll}(v)\ \{\lambda\_.\ \mathsf{cond}(v, M, \{\!|\,|\!\}) * \mathsf{locked}(\mathsf{L}(v), Wt[v{:=}0], Ot)\}$$

Figure 13 New proof rules verifying deadlock-freedom of monitors allowing transferring obligations through notifications, where the parameter M' in the permission cond of a condition variable denotes the obligations which are transferred from the thread notifying that condition variable to the one notified.

if the invariant of l is properly defined such that for any CV v' associated with l and any Wt' and Ot', we have $\mathsf{I}(l)(Wt, Ot) \Rightarrow \mathsf{enoughObs}(v', Wt, Ot)$. Lastly the precondition of this command makes sure that v is associated with lock l, which is enforced by $\mathsf{L}(v){=}l$. 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. Unlike the Leino *et al.* approach [33], this approach has no notion of credits and 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 ChargeObligation and DischargeObligation. 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 invariant enoughObs in the precondition of the rule DischargeObligation.

3.2 Transferring Obligations Through Notifications

In this subsection we extend the Hamin *et al.* approach [15] such that it also allows obligations to be transferred from the notifying thread to the one notified. To this end we make sure that 1) when a thread notifies a CV v the transferred obligations of v, which are transferred through a notification on v, are discharged from the thread only if there is a thread waiting for v which is notified, that is $0{<}Wt(v)$, 2) when a thread waits for a CV v the transferred obligations of v are loaded onto the bag of obligations of that thread as that thread is notified, and 3) when a thread waits for a CV v the level of the lock associated with v is lower than the levels of the transferred obligations of v, too, since when the thread wakes up, where these obligations are loaded onto the bag of the obligations of the thread, this lock must be reacquired. For the sake of simplicity, any CV v on which $\mathsf{notifyAll}(v)$ is performed must have no transferred obligations. The proof rules which need to be updated are shown in Figure 13.

```
routine new_mutex(){
 l := new_lock;
 v := new_cvar;
 mutex(l:=l, v:=v,
   b:=new_bool(false))
}

routine enter_cs(mutex m){
 acquire(m.l);
 while(m.b)
   wait(m.v, m.l);
 m.b := true;
 release(m.l)}

routine exit_cs(mutex m){
 acquire(m.l);
 m.b := false;
 notify(m.v);
 release(m.l)
}
```

Figure 14 Mutexes

```
routine main(){
 m := new_mutex();
 fork(
   while(1){
     enter_cs(m); /* CS */
     exit_cs(m)});
 enter_cs(m); /* CS */
 exit_cs(m)}
```

Figure 15 The main thread is starved if it is scheduled only when the forked thread is in the CS.

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 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. Note that this invariant holds even when some obligations are transferred because these obligations are immediately transferred from the notifying thread to the one notified. 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 invariant 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}.

```
routine new_mutex(){
 l := new_lock;
 v := new_cvar;
 mutex(l:=l, v:=v,
   b:=new_bool(false),
   w:=new_int(0))
}

routine enter_cs(mutex m){
 acquire(m.l);
 if(m.b){
   m.w := m.w+1;
   wait(m.v, m.l)}
 else
   m.b := true;
 release(m.l)}

routine exit_cs(mutex m){
 acquire(m.l);
 m.b := false;
 if(0<m.w){
   m.w := m.w−1;
   m.b := true;
   notify(m.v)}
 release(m.l)}
```

Figure 16 Fair mutexes on top of fair monitors

$\mathsf{mutex}(\mathsf{mutex}\ m, \mathsf{waitobj}\ o) = \mathsf{lock}(m.l) * \mathsf{cond}(m.v, \mathsf{true}, \{m.v\}) \wedge$
$\mathsf{I}(m.l){=}\mathsf{linv}(m) \wedge \mathsf{L}(m.v){=}m.l \wedge \mathsf{R}(m.l) < \mathsf{R}(m.v) \wedge o{=}m.v$

$\mathsf{linv}(\mathsf{mutex}\ m) =$
$\lambda Wt.\ \lambda Ot.\ \exists b, w.\ m.b \mapsto b * m.w \mapsto w \wedge\ Wt(m.v){=}w \wedge (b\ ?\ 0 < Ot(v) : Wt(v) = 0)$

routine new_mutex(){
req : {true}
l := new_lock;
{ulock(l, {}, {}) $\wedge$ R(l)=$r-1$}
v := new_cvar; g_initc(v);
{ulock(l, {}, {}) $*$ cond(v, true, {v}) $\wedge$ R(v)=r $\wedge$ L(v)=l}
m := mutex(l:=l, v:=v, b:=new_bool(false), w:=new_int(0)); g_initl(l); m
ens : {$\lambda m.$ $\exists o.$ mutex(m, o) $\wedge$ R(o)=r}}

routine enter_cs(mutex m){
req : {obs(O) $*$ mutex(m, o) $\wedge$ $o \prec O$}
acquire($m.l$);
{obs($O \uplus \{m.l\}$) $*$ mutex(m, o) $*$ $\exists Wt, Ot.$ locked($m.l$, Wt, Ot) $*$ linv(m)(Wt, Ot)}
if($m.b$){
  $m.w$:= $m.w{+}1$;
  {obs($O \uplus \{m.l\}$) $*$ mutex(m, o) $*$ $\exists Wt, Ot.$ locked($m.l$, Wt, Ot) $*$ linv(m)($Wt \uplus \{m.v\}, Ot$)}
  wait($m.v, m.l$)
  {obs($O \uplus \{m.l, m.v\}$) $*$ mutex(m, o) $*$ $\exists Wt, Ot.$ locked($m.l$, Wt, Ot) $*$ linv(m)(Wt, Ot)}}
else{
  $m.b$:= true; g_chrg($m.v$)
  {obs($O \uplus \{m.l, m.v\}$) $*$ mutex(m, o) $*$ $\exists Wt, Ot.$ locked($m.l$, Wt, Ot) $*$ linv(m)(Wt, Ot)}
};
{obs($O \uplus \{m.l, m.v\}$) $*$ mutex(m, o) $*$ $\exists Wt, Ot.$ locked($m.l$, Wt, Ot) $*$ linv(m)(Wt, Ot)}
release($m.l$)
ens : {obs($O \uplus \{o\}$) $*$ mutex(m, o)}}

routine exit_cs(mutex m){
req : {obs($O \uplus \{o\}$) $*$ mutex(m, o) $\wedge$ $o \prec O$}
acquire($m.l$);
{obs($O \uplus \{m.v, m.l\}$) $*$ mutex(m, o) $*$ $\exists Wt, Ot.$ locked($m.l$, Wt, Ot) $*$ linv(m)(Wt, Ot)}
$m.b$:= false;
if($0{<}m.w$){
  $m.w$:= $m.w{-}1$; $m.b$:= true; notify($m.v$)
  {obs($O \uplus \{m.l\}$) $*$ mutex(m, o) $*$ $\exists Wt, Ot.$ locked($m.l$, Wt, Ot) $*$ linv(m)(Wt, Ot)}}
else{
  g_disch($m.v$)
  {obs($O \uplus \{m.l\}$) $*$ mutex(m, o) $*$ $\exists Wt, Ot.$ locked($m.l$, Wt, Ot) $*$ linv(m)(Wt, Ot)}};
release($m.l$)
ens : {obs(O) $*$ mutex(m, o)}}

Figure 17 Verification of the fair mutexes implementation shown in Figure 16

3.3 Fair Mutexes

In this section we show how our extension helps to verify a fair implementation of a mutex, in which threads are synchronized by CVs. Before introducing this implementation, consider a simple (unfair) implementation of a mutex, shown in Figure 14. In this implementation a mutex consists of a boolean variable b, indicating whether the critical section (CS) is executed by any thread or not, a lock l, protecting this variable from concurrent accesses, and a CV v, preventing threads from entering the CS if there is a thread running that CS. As shown in the routine enter_cs, when a thread tries to enter a CS, protected by a mutex m, it first acquires the mutex's lock and while there is a thread running the CS it releases that lock and suspends itself. If that thread is notified (and reacquires the mutex's lock) while there is no thread running the CS it changes the value of the variable b to true, preventing other threads from entering the CS, and releases the mutex's lock. Before leaving the critical section, as shown in routine exit_cs, this thread acquires the mutex's lock, changes the value of the variable b to false, allowing other threads to enter the CS, notifies the condition variable of the mutex, waking up a waiting thread, if any, and finally releases the mutex's lock.

However, one problem with this implementation is that a thread might be in *starvation*; it might infinitely wait for entering the CS. For example, consider the program in Figure 15, where the main thread is starved if it is scheduled only when the forked thread is in the CS. In this situation, when the forked thread exits the CS, since $m.b$=false, this thread without waiting for $m.v$ again changes $m.b$ to true and enters the CS. To address this problem a new variable w can be added to the structure of the mutex, tracking the number of threads waiting for that mutex. Introducing this variable, the operations enter_cs and exit_cs can be updated, as shown in Figure 16. As shown in the routine enter_cs(m), when a thread tries to enter a CS, if the CS is currently executed by another thread ($m.b$=true) this thread increases the variable $m.w$ and waits for a notification on $m.v$. Otherwise, this threads changes $m.b$ to true and continue its execution. As shown in the routine exit_cs(m), when a thread leaves a CS it first changes $m.b$ to false and if there is a thread waiting for this CS it decreases the number of the waiting threads, changes $m.b$ to true and notifies a thread waiting for $m.v$. Having this implementation, the forked thread in Figure 15 cannot enter the critical section if the main thread is already waiting to enter. Note that this mutex is fair under the assumption that the monitor primitives are fair, i.e. the lock and condition variable wait queues are FIFO.

This implementation can be verified against the expected specifications, shown in Figure 17, if an obligation of $m.v$ is transferred from the thread leaving the CS to the next thread entering the CS. This transfer is necessary because one of the desired invariants of such program is $m.b \Rightarrow 0 < Ot(m.v)$, that is if the CS is executed by any thread there exists an obligation of $m.v$ in the system. Since in the fair implementation of exit_cs before notifying $m.v$ the variable $m.b$ is changed to true, an obligation of $m.v$ must be loaded onto the system. This obligation can be transferred to the notified thread, which is going to enter the CS. Note that this transfer is sound since it is only performed when $0 < m.w$, that is there is a thread waiting for $m.v$ which immediately receives the transferred obligation.

3.4 Fair Readers-Writers Locks

In this section we show how our extension makes it possible to verify a fair implementation of a readers-writers lock, which is synchronized by CVs. Before that, consider a naive implementation of a readers-writers lock (writers-preference), shown in Figure 18, which can be verified by the Hamin *et al.* [15] approach. This lock consists of three variables

```
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,
   aw := new_int(0), ww := new_int(0),
   ar := new_int(0))}

routine acquire_write(rdwr b){
 acquire(b.l);
 while(b.aw+b.ar>0){
   b.ww := b.ww+1;
   wait(b.v_w, b.l);
   b.ww := b.ww−1 };
 b.aw := b.aw+1;
 release(b.l)}

routine release_write(rdwr b){
 acquire(b.l);
 b.aw := b.aw−1;
 notify(b.v_w);
 if(b.ww = 0)
   notifyAll(b.v_r);
 release(b.l)}

routine acquire_read(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)}

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

routine main(){
 rw := new_rdwr();
 fork(
   while(1){
     acquire_read(rw); /* reading ... */
     release_read(rw)
   );
 while(1){
   acquire_write(rw); /* writing ... */
   release_write(rw)
 }}
```

Figure 18 Readers-writers locks synchronized by condition variables

aw, ww, and ar, keeping track of the total number of active writers, waiting writers, and active readers respectively, a lock l, protecting these variables from concurrent accesses, a condition variable v_w, preventing writers from writing while other threads are reading or writing, and a condition variable v_r, preventing readers from reading while there is another thread writing or waiting to write.

However, in this implementation due to the same reason mentioned in the previous section a writer might be starved by other writers, where for example a writer continuously releases the writing lock and without waiting for the CV v_w immediately reacquires it.

A solution to solve this problem is that when a writer releases the writing lock, if there is a waiting writer, in addition to notifying that waiting writer, it also increases the number of active writers. In other words, if there is a waiting writer in the system, the number of the active writers is increased in advance by the thread releasing the writing lock (or the reading lock) and not by the thread acquiring the writing lock. A fair implementation of the readers-writers lock (writers-preference) following this idea along with the details of the specifications of each routine is shown in Figure 19[6]. When a readers-writers lock b is created, a permission $\mathsf{rw}(b, O_w, O_r)$ is provided, where O_w and O_r are two bags of waitable objects whose levels are in an arbitrary client-specified range R. A thread can acquire a writing lock of b if the levels of objects in O_w are lower than the levels of all obligations of that thread. When that lock is acquired the objects in O_w are loaded onto the bag of the obligations of

[6] We inserted $\mathsf{assert}(e)$ commands, shorthand for $\mathsf{while}(\neg e)\{\}$ (loop forever if e is false), to simplify the proof. They can be eliminated using ghost state [27]. The verification of this program without using the assert commands can be found in [17].

$\mathsf{rw}(\mathsf{rdwr}\ b, \mathsf{bag}\langle\mathsf{waitobj}\rangle\ O_w, \mathsf{bag}\langle\mathsf{waitobj}\rangle\ O_r) =$
$\mathsf{lock}(b.l) * \mathsf{cond}(b.v_r, \mathsf{true}, \{\!\}) * \mathsf{cond}(b.v_w, \mathsf{true}, \{\!|b.v_w|\!\}) \wedge \mathsf{I}(ch.l){=}\mathsf{linv}(b) \wedge$
$\mathsf{R}(b.l) < \mathsf{R}(b.v_w) < \mathsf{R}(b.v_r) \wedge O_w{=}\{\!|b.v_r, b.v_w|\!\} \wedge O_r{=}\{\!|b.v_w|\!\}$

$\mathsf{linv}(\mathsf{rdwr}\ b) =$
$\lambda Wt.\ \lambda Ot.\ \exists ar, aw, ww.\ b.ar \mapsto ar * b.aw \mapsto aw * b.ww \mapsto ww \wedge \mathsf{L}(b.v_r){=}\mathsf{L}(b.v_w){=}b.l \wedge$
$0 \leqslant ar \wedge 0 \leqslant aw \wedge 0 \leqslant ww \wedge Wt(b.v_w){=}ww \wedge$
$ar + aw \leqslant Ot(b.v_w) \wedge (Wt(b.v_w) = 0 \vee 0 < ar + aw) \wedge$
$aw + ww \leqslant Ot(b.v_r) \wedge (Wt(b.v_r) = 0 \vee 0 < aw + ww)$

routine new_rdwr()
req : $\{R \cong_< \mathbb{Q}\}$
ens : $\{\lambda b.\ \exists O_w, O_r.\ \mathsf{rw}(b, O_w, O_r) \wedge$
$\mathsf{levels}(O_w) \subseteq R \wedge \mathsf{levels}(O_r) \subseteq R\}$
{l := new_lock();
v_w := new_cvar; g_initc(v_w);
v_r := new_cvar; g_initc(v_r);
b := rdwr($l{:=}l, v_w{:=}v_w, v_r{:=}v_r,$
$aw{:=}$new_int(0), $ww{:=}$new_int(0),
$ar{:=}$new_int(0)); g_initl(l);
b}

routine acquire_write(rdwr b)
req : $\{\mathsf{obs}(O) * \mathsf{rw}(b, O_w, O_r) \wedge O_w \prec O\}$
ens : $\{\mathsf{obs}(O \uplus O_w) * \mathsf{rw}(b, O_w, O_r)\}$
{acquire($b.l$); g_chrg($b.v_r$);
if($b.aw{+}b.ar{>}0$){
$b.ww := b.ww{+}1$;
wait($b.v_w, b.l$)
}
else{ $b.aw := b.aw{+}1$; g_chrg($b.v_w$)};
release($b.l$)}

routine release_write(rdwr b)
req : $\{\mathsf{obs}(O \uplus O_w) * \mathsf{rw}(b, O_w, O_r) \wedge O_w \prec O\}$
ens : $\{\mathsf{obs}(O) * \mathsf{rw}(b, O_w, O_r)\}$
{acquire($b.l$);
assert($0 < b.aw$);
$b.aw := b.aw{-}1$;
if($b.ww > 0$){
notify($b.v_w$);
$b.ww := b.ww{-}1$;
$b.aw := b.aw{+}1$
}
else{ notifyAll($b.v_r$); g_disch($b.v_w$)};
g_disch($b.v_r$);
release($b.l$)}

routine acquire_read(rdwr b)
req : $\{\mathsf{obs}(O) * \mathsf{rw}(b, O_w, O_r) \wedge O_r \prec O\}$
ens : $\{\mathsf{obs}(O \uplus O_r) * \mathsf{rw}(b, O_w, O_r)\}$
{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($b.l$)}

routine release_read(rdwr b)
req : $\{\mathsf{obs}(O \uplus O_r) * \mathsf{rw}(b, O_w, O_r) \wedge O_r \prec O\}$
ens : $\{\mathsf{obs}(O) * \mathsf{rw}(b, O_w, O_r)\}$
{acquire($b.l$);
assert($0 < b.ar$);
$b.ar := b.ar{-}1$;
if($b.ar = 0 \wedge b.ww > 0$){
notify($b.v_w$);
$b.ww := b.ww{-}1$;
$b.aw := b.aw{+}1$
} else g_disch($b.v_w$);
release($b.l$)}

routine main()
req : $\{\mathsf{obs}(\{\!\})\}$
ens : $\{\mathsf{obs}(\{\!\})\}$
{rw := new_rdwr();
fork(
while(1){
acquire_read(rw); /* reading ... */
release_read(rw)
}
);
while(1){
acquire_write(rw); /* writing ... */
release_write(rw)
}
}

Figure 19 Fair readers-writers locks on top of fair monitors, where $R \cong_< \mathbb{Q}$ indicates that R is order-isomorphic to the rational numbers.

this thread, which are discharged when this thread releases this lock. Similar to a writing lock, a reading lock of b can be acquired if the levels of the objects in O_r are lower than the levels of the obligations of the reading thread. When this lock is acquired the objects in O_r are loaded which are discharged when this lock is released. Note that in this program one of the desired invariants is $b.ar + b.aw \leqslant Ot(b.v_w)$. Accordingly, since the variable tracking the number of active writers (aw) is increased in the thread notifying v_w and not in the notified one, it is necessary to load an obligation of v_w onto the notifying thread and then transfer this obligation to the notified thread, through the notification.

4 Related Work

Permissions

One common approach to prove safety properties of a program is assigning permissions to the threads of that program and to make sure that each thread only performs actions for which that thread has permissions [3]. This approach has been adopted by concurrent separation logic [37], where a thread can access a heap location only if it owns that location. This logic has been extended to handle dynamic thread creation [11, 19, 13], rely/guarantee [48, 8], reentrant locking [12], and channels [18, 40].

Jung *et al.* [27] proposed a concurrent separation logic, namely *Iris*, for reasoning about safety of concurrent programs, as the logic in logical relations, and to reason about type systems and data abstraction, among other things. In this logic user-defined protocols on shared state are expressed through *partial commutative monoids* and are enforced through *invariants*. However, the Iris program logic and many other logics such as [41, 36] 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. It follows that these logics do not support programs that use primitive blocking operations. Recently, an extension of Iris, namely *Iron* [1], exploits a notion of obligation to prove absence of resource leaks, but not deadlock-freedom. The adequacy theorem of this logic only considers the state reached by a program after it is completely finished (i.e. all threads have reduced to a value), and it proves that in that state all resources have been freed.

Deadlock

Several approaches to verify termination [35, 14, 43], total correctness [4], and lock-freedom [20] 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 or channels, where the suspension of a notifying/sending thread might cause a waiting thread to be infinitely blocked. In [39] 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 some other works on verifying deadlock-freedom and starvation-freedom of concurrent objects with partial methods, which do not return under certain circumstances

such as acquiring a held lock [34]. In addition to locks these approaches allows to verify other concurrent objects such as sets, stacks and queues. However, this approach is not applicable to condition variables because of *lost notifications*, i.e. a notification on a condition variable is lost if no thread is waiting for that condition variable. Note that releasing a lock, pushing an item into stack, and enqueueing an item when there is no thread waiting for the related concurrent object is not lost, since the next thread, which tries to acquire/pop/dequeue the concurrent object, will not be blocked.

There are also some other approaches addressing some common synchronization bugs of programs in the presence of condition variables. In [49], for example, an approach to identify some potential problems of concurrent programs using condition variables 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 after the notifying thread, that might lead the waiting thread to be infinitely suspended. [28] 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, [6, 9] 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.

There are several verification techniques and type systems to check deadlock-freedom of programs that either synchronize via locks [10, 32, 47] or communicate via messages [7, 30]. Kobayashi [30, 29] 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. In his type system, which works in the context of π-calculus, the send operation is synchronous; i.e. the thread sending on a channel is suspended until the sent message is received by another thread. However, this approach and other approaches, such as [31, 5, 38], verifying deadlock-freedom in the context of π-calculus are not straight forwardly applicable to imperative programming languages.

Obligations

Inspired by the notion of capabilities [30, 29] and implicit dynamic frames [46, 44, 45], Leino *et al.* [33] later integrated deadlock prevention into a verification system for an object-oriented and imperative programing 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. This approach supports asynchronous send operations, where sending on a channel does not suspend the sender thread and there might be a state where a message is sent but not received by any thread. The notion of obligations is used in other verification approaches, which verify deadlock-freedom of semaphores [21], monitors [15, 16] and channels in a separation logic-based system [23, 24], and finite blocking in non-

terminating programs [2]. However, these approaches allow obligations to be transferred only when a thread is forked. In other words, unlike permissions which can be transfered through synchronizations, transferring obligations through synchronizations in these approaches is forbidden. In this paper we provide two mechanisms that allow these approaches to transfer obligations, along with permissions, through synchronization. Our approach can be used to verify deadlock-freedom of imperative programs where some obligation must be transferred through notifications and channels (with asynchronous send operations).

5 Conclusion

This paper introduces two techniques to transfer obligations through synchronization, while ensuring that there is no state where the transferred obligations are lost, i.e. where they are discharged from the sender thread and not loaded onto the receiver thread yet. These techniques allow the obligation-based verification approaches, which modularly verify deadlock-freedom and liveness properties of programs, to transfer obligations, along with permissions, between threads, enabling them to verify a wider range of interesting programs, where obligations must be transferred through synchronizations. We encoded the proposed proof rules in the VeriFast program verifier and succeeded in verifying deadlock-freedom of a number of interesting programs, such as some variations of client-server programs, fair readers-writers locks and dining philosophers, which cannot be modularly verified without such transfer. Integrating the two mechanisms introduced in this paper is an area of future work. Additionally, designing a new variant of Iris for programs with primitive blocking constructs on top of which our presented approach can be built is another important area of future work.

References

1 Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. Iron: Managing obligations in higher-order concurrent separation logic. *To appear in POPL 2019: ACM SIGPLAN Symposium on Principles of Programming Languages, Lissabon, Portugal*, 2019.

2 Pontus Boström and Peter Müller. *Modular verification of finite blocking in non-terminating programs*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

3 John Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.

4 Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular termination verification for non-blocking concurrency. In *ESOP*, pages 176–201, 2016.

5 Ornela Dardha and Simon J Gay. A new linear logic for deadlock-free session-typed processes. In *International Conference on Foundations of Software Science and Computation Structures*, pages 91–109. Springer, 2018.

6 Pedro de Carvalho Gomes, Dilian Gurov, and Marieke Huisman. Specification and verification of synchronization with condition variables. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 3–19. Springer, 2016.

7 Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *European Conference on Object-Oriented Programming*, pages 328–352. Springer, 2006.

8 Xinyu Feng. Local rely-guarantee reasoning. In *ACM SIGPLAN Notices*, volume 44, pages 315–327. ACM, 2009.

9 Pedro de C Gomes, Dilian Gurov, Marieke Huisman, and Cyrille Artho. Specification and verification of synchronization with condition variables. *Science of Computer Programming*, 163:174–189, 2018.

10 Colin S Gordon, Michael D Ernst, and Dan Grossman. Static lock capabilities for deadlock freedom. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 67–78. ACM, 2012.

11 Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Asian Symposium on Programming Languages And Systems*, pages 19–37. Springer, 2007.

12 Christian Haack, Marieke Huisman, and Clément Hurlin. Reasoning about java's reentrant locks. In *Asian Symposium on Programming Languages And Systems*, pages 171–187. Springer, 2008.

13 Christian Haack and Clément Hurlin. Separation logic contracts for a java-like language with fork/join. In *International Conference on Algebraic Methodology and Software Technology*, pages 199–215. Springer, 2008.

14 Jafar Hamin and Bart Jacobs. Modular verification of termination and execution time bounds using separation logic. In *Information Reuse and Integration (IRI), 2016 IEEE 17th International Conference on*, pages 110–117. IEEE, 2016.

15 Jafar Hamin and Bart Jacobs. Deadlock-free monitors. In *European Symposium on Programming*, pages 415–441. Springer, 2018.

16 Jafar Hamin and Bart Jacobs. 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.

17 Jafar Hamin and Bart Jacobs. Deadlock-free monitors and channels. zenodo, `http://doi.org/10.5281/zenodo.3241454`, 2019. URL: `http://doi.org/10.5281/zenodo.3241454`, `doi:10.5281/zenodo.3241454`.

18 Tony Hoare and Peter O'Hearn. Separation logic semantics for communicating processes. *Electronic Notes in Theoretical Computer Science*, 212:3–25, 2008.

19 Aquinas Hobor, Andrew W Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming*, pages 353–367. Springer, 2008.

20 Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 124–133. IEEE, 2013.

21 Bart Jacobs. Provably live exception handling. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*, page 7. ACM, 2015.

22 Bart Jacobs. Verifast 18.02. zenodo, `http://doi.org/10.5281/zenodo.1182724`, 2018. URL: `http://doi.org/10.5281/zenodo.1182724`, `doi:10.5281/zenodo.1182724`.

23 Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

24 Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification of single-threaded and multithreaded programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(3):12, 2018.

25 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. *NASA Formal Methods*, 6617:41–55, 2011.

26 Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. *Programming Languages and Systems*, pages 304–311, 2010.

27 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices*, 50(1):637–650, 2015.

28 Krishna M Kavi, Alireza Moshtaghi, and Deng-Jyi Chen. Modeling multithreaded applications using petri nets. *International Journal of Parallel Programming*, 30(5):353–371, 2002.

29 Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

30 Naoki Kobayashi. A new type system for deadlock-free processes. In *International Conference on Concurrency Theory*, pages 233–247. Springer, 2006.

31 Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Information and Computation*, 252:48–70, 2017.

32 Duy-Khanh Le, Wei-Ngan Chin, and Yong-Meng Teo. An expressive framework for verifying deadlock freedom. In *International Symposium on Automated Technology for Verification and Analysis*, pages 287–302. Springer, 2013.

33 K Rustan M Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *European Symposium on Programming*, pages 407–426. Springer, 2010.

34 Hongjin Liang and Xinyu Feng. Progress of concurrent objects with partial methods. *Proceedings of the ACM on Programming Languages*, 2(POPL):20, 2017.

35 Hongjin Liang, Xinyu Feng, and Zhong Shao. 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)*, page 65. ACM, 2014.

36 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *European Symposium on Programming Languages and Systems*, pages 290–310. Springer, 2014.

37 Peter W O'Hearn. Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1-3):271–307, 2007.

38 Luca Padovani. Type-based analysis of linear communications. *Behavioural Types: from Theory to Tools*, page 193, 2017.

39 Corneliu Popeea and Andrey Rybalchenko. Compositional termination proofs for multi-threaded programs. In *TACAS*, volume 12, pages 237–251. Springer, 2012.

40 David Pym and Chris Tofts. A calculus and logic of resources and processes. *Formal Aspects of Computing*, 18(4):495–517, 2006.

41 Azalea Raad, Jules Villard, and Philippa Gardner. Colosl: Concurrent local subjective logic. In *European Symposium on Programming Languages and Systems*, pages 710–735. Springer, 2015.

42 John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

43 Reuben NS Rowe and James Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 53–65. ACM, 2017.

44 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *Proceedings of the 10th ECOOP Workshop on Formal Techniques for Java-like Programs*, pages 1–12, 2008.

45 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming*, pages 148–172. Springer, 2009.

46 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):2, 2012.

47 Kohei Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Asian Symposium on Programming Languages and Systems*, pages 155–170. Springer, 2008.

48 Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *International Conference on Concurrency Theory*, pages 256–271. Springer, 2007.

49 Chao Wang and Kevin Hoang. Precisely deciding control state reachability in concurrent traces with limited observability. In *VMCAI*, pages 376–394. Springer, 2014.

A Proof of Conditional Server Channels

In this section the verification of the program shown in Figure 11 is illustrated in Figures 20 and 21.

B Proof of Dining Philosophers

Transferring obligations through notifications allows to verify some other interesting programs such as some variants of *dining philosophers*, where a number of philosophers sit at a round table, think, get hungry, and eat if their neighbors are not eating (due to the limited number of forks). An implementation of this program is shown in Figures 22[7], and 23, where a dining_philosophers structure consists of phs, a circular doubly linked list of philosopher, $size$, the size of this list, and a lock (l), where a philosopher structure consists of pre, a pointer to the previous philosopher, $next$, a pointer to the next philosopher, a condition variable v, and a state which can be Thinking, Hungry, or Eating. Calling new_dining_philosophers($size$) creates $size$ philosophers which are initially thinking. Calling pickup(dp, i) makes the ith philosopher in dp start eating if he/she is already hungry and none of its neighbors are eating. Calling putdown(dp, i) makes the ith philosopher in dp stop eating if he/she is already eating, and also makes this philosopher's neighbors eat if they are hungry and their neighbors are not eating.

The specification of the routine new_dining_philosophers in this figure indicates that creating a number of dining philosophers dp produces a permission dp(dp, cvs), where cvs is a list of condition variables associated with philosophers whose levels are assigned arbitrarily. A philosopher can try to start eating only if the level of the CV associated with that philosopher is lower than the levels of the obligations of the running thread. When this philosopher starts eating the CV associated with his/her left and right neighbors are loaded onto the bag of the obligations of the running thread. These obligations are discharged when this philosopher stops eating.

One desired invariant in this program is that the number of obligations for a CV associated with a philosopher ph is greater than the number of his/her neighbors which are eating, that is st($pre.state$) + st($next.state$) $\leqslant$ $Ot(ph.v)$, where st(Eating)=1 and st(Thinking)=st(Hungry)=0. Since the state of a hungry philosopher ph waiting in a suspended thread t is changed to Eating in a thread t' where its neighbor stops eating, an obligation of $ph.v$ must be loaded onto the bag of the obligations of t' and be transferred to t as t' notifies t. This requires transferring obligations through notifications which is possible using our proposed extension. Accordingly, as shown in Figure 22, this program can be verified if for any condition variable v of a philosopher ph, when v is notified the obligations $\{ph.pre.v, ph.next.v\}$ are transferred.

C Transferring Obligations Through Channels: Soundness Proof

In this appendix we provide a formalization and soundness proof for the approach introduced in Section 2. However, unfortunately, there are a few technical differences between this formalization and the one proposed in Section 2 such that in this formalization the ghost information, such as level and transferred permissions and obligations, are associated with

[7] For simplicity, in the proof of this program we avoid writing the heap ownership permissions

$\mathsf{Mch}' ::= \lambda m.\ \mathsf{channel}(\mathsf{snd}(m), \mathsf{Mch}, \mathsf{M}'\mathsf{ch}) * \mathsf{trandit}(\mathsf{snd}(m)) \wedge \mathsf{R}(\mathsf{snd}(m))=1 \wedge$
$\mathsf{M}^{\mathsf{r}}(\mathsf{snd}(m))=\{\!|1|\!\}$
$\mathsf{M}'\mathsf{ch}' ::= \lambda m.\ \{\!|\mathsf{snd}(m)|\!\}$

$\mathsf{Mch} ::= \lambda m.\ \mathsf{channel}(\mathsf{snd}(m), \mathsf{Mch}', \mathsf{M}'\mathsf{ch}') * (\mathsf{fst}(m)=\mathsf{done}\ ?\ \mathsf{true} : \mathsf{trandit}(\mathsf{snd}(m))) \wedge$
$\mathsf{M}^{\mathsf{r}}(\mathsf{snd}(m))=\{\!|1|\!\} \wedge \neg\mathsf{S}(\mathsf{snd}(m))$
$\mathsf{M}'\mathsf{ch} ::= \lambda m.\ \mathsf{fst}(m)=\mathsf{done}\ ?\ \{\!|\}\!| : \{\!|\mathsf{snd}(m)|\!\}$

$\mathsf{Ms} ::= \lambda m.\ \mathsf{channel}(\mathsf{snd}(m), \mathsf{Mch}', \mathsf{M}'\mathsf{ch}') * \mathsf{trandit}(\mathsf{snd}(m)) \wedge \mathsf{M}^{\mathsf{r}}(\mathsf{snd}(m))=\{\!|1|\!\} \wedge \neg\mathsf{S}(\mathsf{snd}(m))$
$\mathsf{M}'\mathsf{s} ::= \lambda m.\ \{\!|\mathsf{snd}(m)|\!\}$

routine server(channel s){
req : $\{\mathsf{obs}(\{\!|\}\!|, \{\!|s^\infty|\!\}) * \mathsf{channel}(s, \mathsf{Ms}, \mathsf{M}'\mathsf{s}) \wedge \mathsf{M}^{\mathsf{r}}(s)=\{\!|1|\!\} \wedge \mathsf{S}(s)\}$
while(true){
inv : $\{\mathsf{obs}(\{\!|\}\!|, \{\!|s^\infty|\!\})\}$
$(thr, ch') := \mathsf{receive}(s);$
$\{\mathsf{obs}(\{\!|ch'|\!\}, \{\!|s^\infty|\!\}) * \mathsf{channel}(ch', \mathsf{Mch}', \mathsf{M}'\mathsf{ch}') * \mathsf{trandit}(ch') \wedge \mathsf{M}^{\mathsf{r}}(ch')=\{\!|1|\!\} \wedge \neg\mathsf{S}(ch')\}$
$ch := \mathsf{new\_channel}();$
$\{\mathsf{obs}(\{\!|ch'|\!\}, \{\!|s^\infty|\!\}) * \mathsf{trandit}(ch') * \mathsf{channel}(ch, \mathsf{Mch}, \mathsf{M}'\mathsf{ch}) \wedge \mathsf{R}(ch)=1 \wedge \mathsf{M}^{\mathsf{r}}(ch)=\{\!|1|\!\} \wedge \neg\mathsf{S}(ch)\}$
$\mathsf{g\_credit}(ch);\ \mathsf{g\_trandit}(ch);$
$\{\mathsf{obs}(\{\!|ch', ch|\!\}, \{\!|s^\infty, ch|\!\}) * \mathsf{trandit}(ch') * \mathsf{credit}(ch) * \mathsf{trandit}(ch)\}$
$\mathsf{send}(ch', (\mathsf{through}, ch));$
$\{\mathsf{obs}(\{\!|\}\!|, \{\!|s^\infty, ch|\!\}) * \mathsf{credit}(ch)\}$
$\mathsf{fork}(\mathsf{cserver}(ch))$
$\{\mathsf{obs}(\{\!|\}\!|, \{\!|s^\infty|\!\})\}$ }
ens : $\{\mathsf{false}\}$}

routine client(channel s){
req : $\{\mathsf{obs}(\{\!|\}\!|, \{\!|\}\!|) * \mathsf{channel}(s, \mathsf{Ms}, \mathsf{M}'\mathsf{s}) * \mathsf{trandit}(s)\}$
$ch' := \mathsf{new\_channel}();$
$\{\mathsf{obs}(\{\!|\}\!|, \{\!|\}\!|) * \mathsf{trandit}(s) * \mathsf{channel}(ch', \mathsf{Mch}', \mathsf{M}'\mathsf{ch}') \wedge \mathsf{R}(ch')=1 \wedge \mathsf{M}^{\mathsf{r}}(ch')=\{\!|1|\!\} \wedge \neg\mathsf{S}(ch')\}$
$\mathsf{g\_credit}(ch');\ \mathsf{g\_trandit}(ch');$
$\{\mathsf{obs}(\{\!|ch'|\!\}, \{\!|ch'|\!\}) * \mathsf{trandit}(s) * \mathsf{credit}(ch') * \mathsf{trandit}(ch')\}$
$\mathsf{send}(s, (\mathsf{through}, ch'));$
$\{\mathsf{obs}(\{\!|\}\!|, \{\!|ch'|\!\}) * \mathsf{credit}(ch')\}$
$(thr, ch) := \mathsf{receive}(ch');\ \mathsf{if}(thr{\neq}\mathsf{through})\ \mathsf{abort}();$
$\{\mathsf{obs}(\{\!|ch|\!\}, \{\!|\}\!|) * \mathsf{channel}(ch, \mathsf{Mch}, \mathsf{M}'\mathsf{ch}) * \mathsf{trandit}(ch) \wedge \mathsf{R}(ch)=1 \wedge \mathsf{M}^{\mathsf{r}}(ch)=\{\!|1|\!\}\}$
$\mathsf{g\_credit}(ch');\ \mathsf{g\_trandit}(ch');$
$\{\mathsf{obs}(\{\!|ch, ch'|\!\}, \{\!|ch'|\!\}) * \mathsf{trandit}(ch) * \mathsf{credit}(ch') * \mathsf{trandit}(ch')\}$
$\mathsf{send}(ch, (\mathsf{request}(), ch'));$
$\{\mathsf{obs}(\{\!|\}\!|, \{\!|ch'|\!\}) * \mathsf{credit}(ch')\}$
$(res, ch_1) := \mathsf{receive}(ch');\ \mathsf{if}(res{=}\mathsf{through} \vee ch{\neq}ch_1)\ \mathsf{abort}();$
$\{\mathsf{obs}(\{\!|ch|\!\}, \{\!|\}\!|) * \mathsf{trandit}(ch)\}$
$\mathsf{send}(ch, (\mathsf{done}, ch'))$
ens : $\{\mathsf{obs}(\{\!|\}\!|, \{\!|\}\!|)\}$}

Figure 20 Verification of the program in Figure 11 (part one of two)

```
routine main(){
req : {obs({}, {})}
 s := new_channel();
 {obs({}, {}) ∗ channel(s, Ms, M's) ∧ M'r(s)={1} ∧ S(s)}
 g_trandits(s);
 {obs({}, {s∞}) ∗ trandit∞(s)}
 fork(
   {obs({}, {s∞})}
   server(s));
 {obs({}, {}) ∗ trandit∞(s)}
 fork(client(s));
 {obs({}, {}) ∗ trandit∞(s)}
 client(s)
ens : {obs({}, {})}}
```

Figure 21 Verification of the program in Figure 11 (part two of two)

channel addresses via the channel permissions rather than via global functions[8]. The proof rules associated with this formalization and the verification of the program in Figure 3, proved using these rules, are shown in Sections C.4 and C.6, respectively.

C.1 Syntax and Semantics of Programs

The syntax of our programming language is defined in Figure 24, where $\mathsf{val}(e)$ is a command that simply yields the value of e as its result and has no side effects, $\mathsf{let}(x, c_1, c_2)$ is syntactic sugar for $x{:=}c_1; c_2$, $\mathsf{fork}(c)$ creates a thread executing c, $\mathsf{while}(c)$ keeps executing c while c evaluates to true, $\mathsf{if}(e, c_1, c_2)$ executes c_1 if e evaluates to true and otherwise it executes c_2, send and receive are used for sending and receiving on channels, and wait, which cannot be used by programmers, indicates that the related thread has tried to receive from an empty channel. Additionally, instead of defining three ghost commands g_credit, g_trandit, and g_trandits, we define a single ghost command nop which is inserted into the program for verification purposes and has no effect on the program's behavior. The expressions used in the syntax of programs can be evaluated and substituted as shown in Figure 25.

The small step semantics, defined in Figure 27, relates two *configurations*, defined in Figure 26. A configuration consists of a heap, which maps a channel identifier to the list of messages of that channel; a thread table, which maps a thread identifier to the pair command-context related to that thread; and a list of server channels.

$$\mathsf{dp}(\text{dining\_philosophers } dp,\ \mathsf{list}\langle\mathsf{waitobj}\rangle\ cvs) =$$
$$\mathsf{lock}(dp.l) * \mathsf{phs\_cvs}(dp.phs, dp.phs.pre, cvs) \wedge$$
$$\mathsf{I}(dp.l){=}\mathsf{linv}(dp) \wedge \mathsf{L}(dp.l){=}\mathsf{new\_dp} \wedge \mathsf{size}(cvs){=}dp.size \wedge \forall 0{\leqslant}i{\leqslant}size.\ \mathsf{R}(r_l) < \mathsf{R}(rs[i])$$

$$\mathsf{phs\_cvs}(ph, ph', cvs) =$$
$$ph{=}ph' \ ?\ cvs{=}[ph.v] :\ \exists cvs',\ cvs{=}[ph.v :: cvs'] * \mathsf{phs\_cvs}(ph.next, ph', cvs')$$

$$\mathsf{linv}(\mathsf{dp}\ dp) = \lambda Wt.\ \lambda Ot.\ \mathsf{philosophers}(dp.phs, dp.phs.pre, Wt, Ot, dp.l)$$

$$\mathsf{philosophers}(ph, ph', Wt, Ot, l) =\ \mathsf{philosopher}(ph, Wt, Ot, l, ph.pre, ph.next) *$$
$$ph{=}ph' \ ?\ \mathsf{true} : \mathsf{philosophers}(ph.next, ph', Wt, Ot, l)$$

$$\mathsf{philosopher}(ph, Wt, Ot, l, pre, next) = \mathsf{cond}(ph.v, \mathsf{true}, \{pre.v, next.v\}) \wedge$$
$$pre{=}ph.pre \wedge next{=}ph.next \wedge ph{\neq}pre \wedge ph{\neq}next \wedge pre{\neq}next \wedge$$
$$next.pre{=}ph \wedge pre.next{=}ph \wedge \mathsf{L}(ph.v){=}l \wedge 0 \leqslant Wt(ph.v) \wedge Wt(ph.v) \leqslant 1 \wedge$$
$$(ph.state{=}\text{Hungry} \vee Wt(ph.v) \leqslant 0) \wedge (ph.state{\neq}\text{Hungry} \vee 0 < Wt(ph.v)) \wedge$$
$$(Wt(ph.v) \leqslant 0 \vee 0 < \mathsf{st}(pre.state) + \mathsf{st}(next.state)) \wedge$$
$$\mathsf{st}(pre.state) + \mathsf{st}(next.state) \leqslant Ot(ph.v)$$

$$\mathsf{st}(state) ::= state{=}\text{Eating}\ ?\ 1 : 0$$

routine new_dining_philosophers(int $size$){
req : $\{\forall 0{\leqslant}i{<}size.\ r_l < rs[i] \wedge \mathsf{size}(rs){=}size \wedge 2 < size\}$
ens : $\{\lambda dp.\ \mathsf{dp}(dp, cvs) \wedge \mathsf{size}(cvs){=}size \wedge \mathsf{R}(dp.l){=}r_l \wedge \forall 0{\leqslant}i{<}size.\ \mathsf{R}(cvs[i]){=}rs[i]\}$
phs := philosopher($state$:=new_cell(Thinking), v:=new_cvar,
  pre:=new_cell(null), $next$:=new_cell(null));
$ph1$:= philosopher($state$:=new_cell(Thinking), v:=new_cvar,
  pre:=new_cell(null), $next$:=new_cell(null));
$ph2$:= philosopher($state$:=new_cell(Thinking), v:=new_cvar,
  pre:=new_cell(null), $next$:=new_cell(null));
$phs.pre := ph2$; $phs.next := ph1$;
$ph1.pre := phs$; $ph1.next := ph2$;
$ph2.pre := ph1$; $phs.next := phs$;
l := new_lock; $i := 3$
while($i{<}size$){
  ph:=philosopher($state$:=new_cell(Thinking), v:=new_cvar, pre:=phs, $next$:=$phs.next$);
  $phs.next.pre := ph$;
  $phs.next = ph$;
  $i := i{+}1$
};
dining_philosophers(phs:=phs, l:=l, $size$:=$size$)}

Figure 22 Dining philosophers (part one of two)

```
routine pickup(dining_philosophers dp, int i){
req : {obs(O) * dp(dp, cvs) ∧ cvs[i] ≺ O ∧ 0⩽i<size(cvs)}
ens : {obs(O⊎{|cvs[(i+n−1)%n], cvs[(i+1)%n]|}) * dp(dp, cvs) ∧ n=size(cvs)}
 acquire(dp.l);
 ph := dp.phs;
 j := new_int(0);
 while(j < i){
   ph := ph.next;
   j := j+1
 };
 if(ph.state ≠ Thinking)
   abort;
 ph.state := Hungry;
 if(ph.next.state=Eating ∨ ph.pre.state=Eating)
   wait(ph.v, dp.l)
 else
   ph.state := Eating;
 release(dp.l)}

routine putdown(dining_philosophers dp, int i){
req : {obs(O) * dp(dp, cvs) ∧ 0⩽i<size(cvs)}
ens : {obs(O−{|cvs[(i+n−1)%n], cvs[(i+1)%n]|}) * dp(dp, cvs) ∧ n=size(cvs)}
 acquire(dp.l);
 ph := dp.phs;
 j := new_int(0);
 while(j < i){
   ph := ph.next;
   j := j+1
 };
 if(ph.state ≠ Eating)
   abort;
 ph.state := Thinking;
 test_and_notify(ph.next);
 test_and_notify(ph.pre);
 release(dp.l)}

routine test_and_notify(philosopher ph)
{if(ph.next.state≠Eating ∧ ph.pre.state≠Eating ∧ ph.state=Hungry){
   ph.state := Eating;
   notify(ph.v)
 }}
```

Figure 23 Dining philosophers (part two of two)

$$
\begin{aligned}
&c \in \mathit{Commands},\ e \in \mathit{Expressions},\ z \in \mathbb{Z},\ b \in \mathit{Booleans},\ x \in \mathit{Variables}\\
&e ::= z \mid x \mid e_1{+}e_2 \mid (e_1, e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e)\\
&\qquad \mid \mathsf{true} \mid e_1 = e_2 \mid e_1 \leqslant e_2 \mid \neg e\\
&c ::= \mathsf{val}(e) \mid \mathsf{let}(x, c_1, c_2) \mid \mathsf{fork}(c) \mid \mathsf{while}(c) \mid \mathsf{if}(e, c_1, c_2)\\
&\qquad \mid \mathsf{new\_channel}(b) \mid \mathsf{send}(e_1, e_2) \mid \mathsf{receive}(e) \mid \mathsf{wait}(e) \mid \mathsf{nop}
\end{aligned}
$$

Figure 24 Syntax of the programming language

C.2 Syntax and Semantics of Assertions

The syntax of assertions is defined in Figure 28[9]. Note that the *location* of a channel ch consists of the *obligation* of ch and the permissions and the obligations which are transferred through a specific message sent on ch, denoted by $\mathsf{M}(ch)$ and $\mathsf{M}'(ch)$. Also note that the permissions which are transferred through a specific message sent on ch, denoted by $\mathsf{M}(ch)$, are specified through an index (as well as the required arguments) pointing to a table in which each element is a function that given a list of arguments and a message returns an assertion. This makes it possible for the predicates specifying these permissions to recursively refer to themselves or to each other, as in Figure 20[10]. The obligation of a location ch, denoted by $\mathsf{O}(ch)$, consists of the address of that location, denoted by $\mathsf{A}(ch)$, as well as other related information such as the level of ch, denoted by $\mathsf{R}(ch)$; the bag of the levels of the obligations which are possibly imported by ch, denoted by $\mathsf{M}^{\mathsf{r}}(ch)$; and whether ch is a server channel or not, denoted by $\mathsf{S}(ch)$.

The proposed assertions describe some ghost resources, namely p, $\tilde{O}$, $\tilde{I}$, C, T, that keep track of allocated channels, and the current thread's obligations, importers, credits, and trandits, respectively, shown in Figure 29.

C.3 Weakest Precondition of Commands

The weakest precondition of a command c for $n{>}0$ steps w.r.t. a postcondition Q (with a given predicate table, specified by pt), denoted by $\mathsf{wp}_{n,pt}(c, Q)$ is defined in Figure 30. Note that $\mathsf{wp}(c, Q)_{0,pt} = \mathsf{true}$. Also note that for the sake of simplicity the index pt is elided. Having this definition, we define the weakest precondition of a context and the weakest precondition of a command-context as shown in Definitions 1 and 2. Having these definitions, we can prove some auxiliary lemmas, shown in Lemmas 4, 5, 6, and 7, which are used to prove Theorem 13.

---

[8] The reason is to make this formalization consistent with the one in Appendix D which is machine-checked with Coq, where ghost information is associated with lock and condition variable addresses via the lock and cond permissions. However, we believe one way to formalize the precise approach of Section 3 would be to define assertions as functions from ghost information to separating conjunctions of chunks. In the soundness proof, one would track these as partial functions whose domain is the set of allocated addresses. The functions passed into the assertions would be *totalizations* of these partial functions. An assertion is true if it is true for all totalizations of the functions.

[9] Note that we use a shallow embedding: assertions have no variables; to model quantifications, we use meta-level functions from values to assertions.

[10] An alternative approach is to use a step-indexed domain of assertions, as in Iris [27]. There, ▶ *Assertions* could be used instead of *Indexes* × *Lists*(*Arguments*), where ▶ is Iris's *guard* for *guarded recursive definitions*.

$v \in \mathit{Values} ::= z \mid b \mid (v, v)$
$[\![]\!] \in \mathit{Expressions} \to \mathit{Values}$

$[\![z]\!]{=}z$
$[\![x]\!]{=}0$
$[\![e_1{+}e_2]\!]{=}[\![e_1]\!]{+}[\![e_2]\!]$
$[\![(e_1, e_2)]\!]{=}([\![e_1]\!], [\![e_2]\!])$
$[\![\mathsf{true}]\!]{=}\mathsf{true}$
$[\![(e_1 = e_2)]\!]{=}([\![e_1]\!] = [\![e_2]\!])$
$[\![(e_1 \leqslant e_2)]\!]{=}([\![e_1]\!] \leqslant [\![e_2]\!])$
$[\![(\neg e)]\!]{=}(\neg [\![e]\!])$
$$[\![\mathsf{fst}(e)]\!] = \begin{cases} [\![e_1]\!] & \text{if } e{=}(e_1, e_2) \\ 0 & \text{otherwise} \end{cases}$$
$$[\![\mathsf{snd}(e)]\!] = \begin{cases} [\![e_2]\!] & \text{if } e{=}(e_1, e_2) \\ 0 & \text{otherwise} \end{cases}$$

$z[v/x] = z$
$x[v/x'] = x \quad \text{if } x \neq x'$
$x[v/x] = v$
$(e_1{+}e_2)[v/x] = e_1[v/x]{+}e_2[v/x]$
$(e_1, e_2)[v/x] = (e_1[v/x], e_2[v/x])$
$\mathsf{true}[v/x] = \mathsf{true}$
$(e_1 = e_2)[v/x] = (e_1[v/x] = e_2[v/x])$
$(e_1 \leqslant e_2)[v/x] = (e_1[v/x] \leqslant e_2[v/x])$
$(\neg e)[v/x] = (\neg e[v/x])$
$\mathsf{fst}(e)[v/x] = \mathsf{fst}(e[v/x])$
$\mathsf{snd}(e)[v/x] = \mathsf{snd}(e[v/x])$

$\mathsf{val}(e)[v/x] = \mathsf{val}(e[v/x])$
$\mathsf{let}(x, c_1, c_2)[v/x] = \mathsf{let}(x, c_1, c_2)$
$\mathsf{let}(x, c_1, c_2)[v/x'] = \mathsf{let}(x, c_1[v/x'], c_2[v/x']) \quad \text{if } x \neq x'$
$\mathsf{fork}(c)[v/x] = \mathsf{fork}(c[v/x])$
$\mathsf{while}(c)[v/x] = \mathsf{while}(c[v/x])$
$\mathsf{if}(e, c_1, c_2)[v/x] = \mathsf{if}(e[v/x], c_1[v/x], c_2[v/x])$
$\mathsf{new\_channel}(b)[v/x] = \mathsf{new\_channel}(b)$
$\mathsf{send}(e_1, e_2)[v/x] = \mathsf{send}(e_1[v/x], e_2[v/x])$
$\mathsf{receive}(e)[v/x] = \mathsf{receive}(e[v/x])$
$\mathsf{nop}[v/x] = \mathsf{nop}$

Figure 25 Evaluation of expressions and substitution of expressions and commands

$$
\begin{array}{l}
m \in \mathit{Messages} = \mathit{Values} \\
\mathit{Addresses} = \mathbb{Z} \\
\mathit{ThreadIds} = \mathbb{Z} \\
h \in \mathit{Heaps} = \mathit{Addresses} \rightharpoonup \mathit{Lists}(\mathit{Messages}) \\
\xi \in \mathit{Contexts} ::= \mathsf{done} \mid \mathsf{let}'(x, c, \xi) \mid \mathsf{while}'(c, \xi) \\
\theta \in \mathit{ThreadConfigurations} ::= (c; \xi) \\
t \in \mathit{ThreadTables} = \mathit{ThreadIds} \rightharpoonup \mathit{ThreadConfigurations} \\
s \in \mathit{ServerChannelsSets} = \mathit{Sets}(\mathit{Addresses}) \\
\kappa \in \mathit{Configurations} = \mathit{Heaps} \times \mathit{ThreadTables} \times \mathit{ServerChannelsSets}
\end{array}
$$

Figure 26 Configurations

$$
\begin{array}{l}
(t[id{:=}\mathsf{new\_channel}(b); \xi], h[z{:=}\varnothing], s) \rightsquigarrow (t[id{:=}\mathsf{val}(z); \xi], h[z{:=}[]], b{=}\mathsf{true}\ ?\ s \cup \{z\} : s) \\
(t[id{:=}\mathsf{send}(e_1, e_2); \xi], h[[\![e_1]\!]{:=}M], s) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h[[\![e_1]\!]{:=}M \cdot [[\![e_2]\!]]], s) \\
(t[id{:=}\mathsf{receive}(e); \xi], h[[\![e]\!]{:=}[m] \cdot M], s) \rightsquigarrow (t[id{:=}\mathsf{val}(m); \xi], h[[\![e]\!]{:=}M], s) \\
(t[id{:=}\mathsf{receive}(e); \xi], h[[\![e]\!]{:=}[]], s) \rightsquigarrow (t[id{:=}\mathsf{wait}(e); \xi], h[[\![e]\!]{:=}[]], s) \\
(t[id{:=}\mathsf{wait}(e); \xi], h[[\![e]\!]{:=}[m] \cdot M], s) \rightsquigarrow (t[id{:=}\mathsf{val}(m); \xi], h[[\![e]\!]{:=}M], s) \\
(t[id{:=}\mathsf{fork}(c); \xi, id'{:=}\varnothing], h, s) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi, id'{:=}c; \mathsf{done}], h, s) \\
(t[id{:=}\mathsf{let}(x, c_1, c_2); \xi], h, s) \rightsquigarrow (t[id{:=}c_1; \mathsf{let}'(x, c_2, \xi)], h, s) \\
(t[id{:=}\mathsf{val}(e); \mathsf{let}'(x, c, \xi)], h, s) \rightsquigarrow (t[id{:=}c[[\![e]\!]/x]; \xi], h, s) \\
(t[id{:=}\mathsf{val}(e); \mathsf{done}], h, s) \rightsquigarrow (t[id{:=}\varnothing], h, s) \\
(t[id{:=}\mathsf{if}(e, c_1, c_2); \xi], h, s) \rightsquigarrow (t[id{:=}c_1; \xi], h, s) \quad \text{if } [[\![e]\!] = \mathsf{true} \\
(t[id{:=}\mathsf{if}(e, c_1, c_2); \xi], h, s) \rightsquigarrow (t[id{:=}c_2; \xi], h, s) \quad \text{if } [[\![e]\!] \neq \mathsf{true} \\
(t[id{:=}\mathsf{while}(c); \xi], h, s) \rightsquigarrow (t[id{:=}c; \mathsf{while}'(c, \xi)], h, s) \\
(t[id{:=}\mathsf{val}(e); \mathsf{while}'(c, \xi)], h, s) \rightsquigarrow (t[id{:=}c; \mathsf{while}'(c, \xi)], h, s) \quad \text{if } [[\![e]\!] = \mathsf{true} \\
(t[id{:=}\mathsf{val}(e); \mathsf{while}'(c, \xi)], h, s) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h, s) \quad \text{if } [[\![e]\!] \neq \mathsf{true} \\
(t[id{:=}\mathsf{nop}; \xi], h, s) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h, s)
\end{array}
$$

Figure 27 Semantics of programs, where tt stands for $\mathsf{val}(0)$, and $[m]$ represents a list with one element m, and $M_1 \cdot M_2$ appends two lists M_1 and M_2.

$n \in \mathbb{N}$
$\mathbb{NF} ::= n \mid \infty$
$\mathit{Bags}(A) = A \to \mathbb{NF}$
$\mathit{Indexes} = \mathbb{Z}$
$\mathit{Arguments} = \mathbb{Z}$
$r \in \mathit{Levels} = \mathbb{R}$
$o \in \mathit{Obligations} = \mathit{Addresses} \times \mathit{Levels} \times \mathit{Bags}(\mathit{Levels}) \times \mathit{Booleans}$
$l \in \mathit{Locations} =$
$\quad \mathit{Obligations} \times (\mathit{Indexes} \times \mathit{Lists}(\mathit{Arguments})), \mathit{Messages} \to \mathit{Bags}(\mathit{Obligations})$
$O, I \in \mathit{Bags}(\mathit{Obligations})$
$b \in \mathit{Booleans}$
$\hat{v} \in \mathit{AValues} ::= z \mid r \mid b \mid l \mid o \mid O$
$\alpha \in \mathit{AValues} \to \mathit{Assertions}$
$a \in \mathit{Assertions} ::= \mathsf{channel}(l) \mid \mathsf{credit}(z) \mid \mathsf{trandit}(z) \mid \mathsf{trandit}^{\infty}(z) \mid \mathsf{obs}(O, I)$
$\qquad\qquad\qquad\qquad \mid b \mid a_1 \wedge a_2 \mid a_1 \vee a_2 \mid a_1 * a_2 \mid a_1 \mathrel{-\!\!*} a_2 \mid \forall\alpha \mid \exists\alpha$

$\mathsf{pt} : \mathit{PredicateTables} = \mathit{Indexes} \to \mathit{Lists}(\mathit{Arguments}) \to \mathit{Messages} \to \mathit{Assertions}$
$\mathsf{O} : \mathit{Locations} \to \mathit{Obligations}$, where $\mathsf{O}((A, R, M^r, S), M, M') = (A, R, M^r, S)$
$\mathsf{A} : \mathit{Locations} \to \mathit{Addresses}$, where $\mathsf{A}((A, R, M^r, S), M, M') = A$
$\mathsf{R} : \mathit{Locations} \to \mathit{Levels}$, where $\mathsf{R}((A, R, M^r, S), M, M') = R$
$\mathsf{M^r} : \mathit{Locations} \to \mathit{Bags}(\mathit{Levels})$, where $\mathsf{M^r}((A, R, M^r, S), M, M') = M^r$
$\mathsf{S} : \mathit{Locations} \to \mathit{Booleans}$, where $\mathsf{S}((A, R, M^r, S), M, M') = S$
$\mathsf{M} : \mathit{Locations} \to \mathit{Messages} \to \mathit{Assertions}$
$\quad$ where $\mathsf{M}((A, R, M^r, S), (M_1, M_2), M') = \mathsf{pt}(M_1, M_2)$
$\mathsf{M'} : \mathit{Locations} \to \mathit{Messages} \to \mathit{Bags}(\mathit{Obligations})$, where $\mathsf{M'}((A, R, M^r, S), M, M') = M'$
$\mathsf{Ro} : \mathit{Obligations} \to \mathit{Levels}$, where $\mathsf{Ro}(A, R, M^r, S) = R$

Figure 28 Syntax of assertions

$p \in \mathit{PermissionHeaps} = \mathit{Locations} \rightharpoonup \{\mathsf{channel}\}$
$\mathit{Option}(A) ::= s \mid \varnothing$, where $s \in A$
$\tilde{O}, \tilde{I} \in \mathit{Option}(\mathit{Bags}(\mathit{Obligations}))$
$C, T \in \mathit{Bags}(\mathit{Addresses})$

$p, \tilde{O}, \tilde{I}, C, T \models \mathsf{channel}(l) \quad \Leftrightarrow \quad p(l) = \mathsf{channel}$
$p, \tilde{O}, \tilde{I}, C, T \models \mathsf{credit}(z) \quad \Leftrightarrow \quad 0 < C(z)$
$p, \tilde{O}, \tilde{I}, C, T \models \mathsf{trandit}(z) \quad \Leftrightarrow \quad 0 < T(z)$
$p, \tilde{O}, \tilde{I}, C, T \models \mathsf{trandit}^{\infty}(z) \quad \Leftrightarrow \quad T(z) = \infty$
$p, \tilde{O}, \tilde{I}, C, T \models \mathsf{obs}(O, I) \quad \Leftrightarrow \quad \tilde{O} = O \wedge \tilde{I} = I$
$p, \tilde{O}, \tilde{I}, C, T \models b \quad \Leftrightarrow \quad b = \mathsf{true}$
$p, \tilde{O}, \tilde{I}, C, T \models a_1 \wedge a_2 \quad \Leftrightarrow \quad p, \tilde{O}, \tilde{I}, C, T \models a_1 \wedge p, \tilde{O}, \tilde{I}, C, T \models a_2$
$p, \tilde{O}, \tilde{I}, C, T \models a_1 \vee a_2 \quad \Leftrightarrow \quad p, \tilde{O}, \tilde{I}, C, T \models a_1 \vee p, \tilde{O}, \tilde{I}, C, T \models a_2$
$p, \tilde{O}, \tilde{I}, C, T \models a_1 * a_2 \quad \Leftrightarrow \quad \exists p_1, p_2, \tilde{O}_1, \tilde{O}_2, \tilde{I}_1, \tilde{I}_2, C_1, C_2, T_1, T_2.$
$p{=}p_1 \uplus p_2 \wedge \tilde{O}{=}\tilde{O}_1 \tilde{\uplus} \tilde{O}_2 \wedge \tilde{I}{=}\tilde{I}_1 \tilde{\uplus} \tilde{I}_2 \wedge C{=}C_1 \uplus C_2 \wedge T{=}T_1 \uplus T_2 \wedge$
$p_1, \tilde{O}_1, \tilde{I}_1, C_1, T_1 \models a_1 \wedge p_2, \tilde{O}_2, \tilde{I}_2, C_2, T_2 \models a_2$
$p, \tilde{O}, \tilde{I}, C, T \models a_1 \mathrel{-\!\!*} a_2 \quad \Leftrightarrow \quad \forall p_1, \tilde{O}_1, \tilde{I}_1, C_1, T_1.\ p_1, \tilde{O}_1, \tilde{I}_1, C_1, T_1 \models a_1 \wedge$
$\tilde{O} \perp \tilde{O}_1 \wedge \tilde{I} \perp \tilde{I}_1 \Rightarrow (p \uplus p_1), (\tilde{O} \tilde{\uplus} \tilde{O}_1), (\tilde{I} \tilde{\uplus} \tilde{I}_1), (C \uplus C_1), (T \uplus T_1) \models a_2$
$p, \tilde{O}, \tilde{I}, C, T \models \forall \alpha \quad \Leftrightarrow \quad \forall \hat{v} {\in} \mathit{AValues}.\ p, \tilde{O}, \tilde{I}, C, T \models \alpha(\hat{v})$
$p, \tilde{O}, \tilde{I}, C, T \models \exists \alpha \quad \Leftrightarrow \quad \exists \hat{v} {\in} \mathit{AValues}.\ p, \tilde{O}, \tilde{I}, C, T \models \alpha(\hat{v})$

$a_1 \vdash a_2 \quad \Leftrightarrow \quad (\forall p, \tilde{O}, \tilde{I}, C, T.\ p, \tilde{O}, \tilde{I}, C, T \models a_1 \Rightarrow p, \tilde{O}, \tilde{I}, C, T \models a_2)$

where for any $p_1, p_2 \in A \rightharpoonup B$ and $\tilde{O}_1, \tilde{O}_2 \in \mathit{Option}(\mathit{Bags}(A))$ and $B_1, B_2 \in \mathit{Bags}(A)$

$\tilde{O}_1 \perp \tilde{O}_2 \quad \Leftrightarrow \quad \tilde{O}_1 = \varnothing \vee \tilde{O}_2 = \varnothing$

$$p_1 \uplus p_2 = \lambda v. \begin{cases} p_1(v) & \text{if } p_2(v){=}\varnothing \\ p_2(v) & \text{otherwise} \end{cases} \qquad \tilde{O}_1 \tilde{\uplus} \tilde{O}_2 = \begin{cases} \tilde{O}_1 & \text{if } \tilde{O}_2{=}\varnothing \\ \tilde{O}_2 & \text{if } \tilde{O}_1{=}\varnothing \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$B_1 \uplus B_2 = \lambda v. \begin{cases} \infty & \text{if } B_1(v){=}\infty \text{ or } B_2(v){=}\infty \\ B_1(v) + B_2(v) & \text{otherwise} \end{cases}$$

Figure 29 Satisfaction relation

$$\mathsf{wp} \in \mathit{WeakestPreconditions} = \mathit{Commands} \to (\mathit{Values} \to \mathit{Assertions}) \to \mathbb{N} \to \mathit{PredicateTables} \to \mathit{Assertions}$$

$$\mathsf{levels}(O) = \{\!|\mathsf{Ro}(o) \mid o \in O|\!\}$$
$$o \prec' R \Leftrightarrow \forall r{\in}R.\ \mathsf{Ro}(o) < r$$
$$o \prec O \Leftrightarrow o \prec' \mathsf{levels}(O)$$
$$o \prec^r I \Leftrightarrow \forall o'{\in}I.\ o{=}o' \vee o \prec' \mathsf{M}^{\mathsf{r}}(o')$$

$$\mathsf{wp}_n(\mathsf{val}(e), Q) = Q([\![e]\!])$$
$$\mathsf{wp}_n(\mathsf{new\_channel}(b), Q) = \forall z.\ \exists r, M^r, M, M'.\ \mathsf{channel}((z, r, M^r, b), M, M') \mathrel{-\!\!*} Q(z)$$
$$\mathsf{wp}_n(\mathsf{send}(e_1, e_2)) = \exists O, I, ch, m.\ (\mathsf{obs}(O, I) * \mathsf{channel}(ch) * \mathsf{M}(ch)(m) *$$
$$(\mathsf{M}^{\mathsf{r}}(ch){=}\{\!|\,|\!\} \vee \mathsf{trandit}(\mathsf{A}(ch))) \wedge \mathsf{levels}(\mathsf{M}'(ch)(m)){\subseteq}\mathsf{M}^{\mathsf{r}}(ch) \wedge \mathsf{A}(ch) = [\![e_1]\!] \wedge m = [\![e_2]\!])$$
$$* ((\mathsf{obs}(O{-}\{\!|\mathsf{O}(ch)|\!\}{-}\mathsf{M}'(ch)(m), I) * \mathsf{channel}(ch)) \mathrel{-\!\!*} Q(\mathsf{tt}))$$
$$\mathsf{wp}_n(\mathsf{receive}(e)) = \exists O, I, ch.\ (\mathsf{obs}(O, I) * \mathsf{channel}(ch) * ((\mathsf{S}(ch) \vee \mathsf{credit}(\mathsf{A}(ch))) \wedge$$
$$\mathsf{O}(ch){\prec}O \wedge \mathsf{O}(ch){\prec^r}I \wedge (\neg\mathsf{S}(ch) \vee (O{=}\{\!|\,|\!\} \wedge \forall o{\in}I.\ o{=}\mathsf{O}(ch))) \wedge \mathsf{A}(ch) = [\![e]\!])) *$$
$$\forall m.\ ((\mathsf{obs}(O{\uplus}\mathsf{M}'(ch)(m), I{-}\{\!|\mathsf{O}(ch)|\!\}) * \mathsf{M}(ch)(m)) \mathrel{-\!\!*} Q(m))$$
$$\mathsf{wp}_n(\mathsf{wait}(e)) = \mathsf{wp}_n(\mathsf{receive}(e))$$
$$\mathsf{wp}_n(\mathsf{fork}(c), Q) = \exists O_1, O_2, I_1, I_2.\ \mathsf{obs}(O_1{\uplus}O_2, I_1{\uplus}I_2) * (\mathsf{obs}(O_1, I_1) \mathrel{-\!\!*} Q(\mathsf{tt})) *$$
$$(\mathsf{obs}(O_2, I_2) \mathrel{-\!\!*} \mathsf{wp}_{n-1}(c, \lambda\_.\ \mathsf{obs}(\{\!|\,|\!\}, \{\!|\,|\!\})))$$
$$\mathsf{wp}_n(\mathsf{if}(e, c_1, c_2), Q) = ([\![e]\!] = \mathsf{true})\ ?\ \mathsf{wp}_{n-1}(c_1, Q) : \mathsf{wp}_{n-1}(c_2, Q)$$
$$\mathsf{wp}_n(\mathsf{while}(c), Q) = \mathsf{wp}_{n-1}(c, (\lambda vl.\ vl{\neq}\mathsf{true}\ ?\ Q(\mathsf{tt}) : \mathsf{wp}_{n-1}(\mathsf{while}(c), Q)))$$
$$\mathsf{wp}_n(\mathsf{nop}) =$$
$$\boxed{\text{as g\_credit}}$$
$$(\exists O, I, ch.\ \mathsf{obs}(O, I) * \mathsf{channel}(ch) *$$
$$((\mathsf{obs}(O{\uplus}\{\!|\mathsf{O}(ch)|\!\}, I) * \mathsf{channel}(ch) * \mathsf{credit}(\mathsf{A}(ch))) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee$$
$$\boxed{\text{as g\_trandit}}$$
$$(\exists O, I, ch.\ \mathsf{obs}(O, I) * \mathsf{channel}(ch) *$$
$$((\mathsf{obs}(O, I{\uplus}\{\!|\mathsf{O}(ch)|\!\}) * \mathsf{channel}(ch) * \mathsf{trandit}(\mathsf{A}(ch))) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee$$
$$\boxed{\text{as g\_trandits}}$$
$$(\exists O, I, ch.\ \mathsf{obs}(O, I) * \mathsf{channel}(ch) *$$
$$((\mathsf{obs}(O, I{\uplus}(\lambda o.\ o{=}\mathsf{O}(ch)\ ?\ \infty : 0)) * \mathsf{channel}(ch) * \mathsf{trandit}^{\infty}(\mathsf{A}(ch))) \mathrel{-\!\!*} Q(\mathsf{tt})))$$

Figure 30 Weakest precondition, where tt stands for 0.

▶ **Definition 1** (Weakest Precondition of a Context).

$$\mathsf{wpx}_n(\xi) = \begin{cases} \lambda\_.\ \mathsf{obs}(\{\!\}, \{\!\}) & \textit{if } \xi{=}\mathsf{done} \\ \lambda v.\ \mathsf{wp}_n(c[v/x], \mathsf{wpx}_n(\xi')) & \textit{if } \xi{=}\mathsf{let}'(x, c, \xi') \\ \lambda v.\ v \neq \mathsf{true}\ ?\ \mathsf{wp}_n(\mathsf{tt}, \mathsf{wpx}_n(\xi')) : \mathsf{wp}_n(c, \mathsf{wpx}_{n-1}(\xi)) & \textit{if } \xi{=}\mathsf{while}'(c, \xi') \end{cases}$$

▶ **Definition 2** (Weakest Precondition of a command-context).

$$\mathsf{wpcx}_n(c, \xi) = \mathsf{wp}_n(c, \mathsf{wpx}_n(\xi))$$

▶ **Lemma 3** (Weakening Postcondition).

$$\forall n, c, Q, Q', p, \tilde{O}, \tilde{I}, C, T.\quad p, \tilde{O}, \tilde{I}, C, T \models \mathsf{wp}_n(c, Q) \land (\forall z.\ Q(z) \vdash Q'(z)) \Rightarrow \\ \forall n' {\leqslant} n.\ p, \tilde{O}, \tilde{I}, C, T \models \mathsf{wp}_{n'}(c, Q')$$

Proof. By induction on n and case analysis of c. ◀

▶ **Lemma 4** (Weakest Precondition of new_channel).

$$\forall b, \xi, p, O, I, C, T. \\ p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{new\_channel}(b), \xi) \Rightarrow \forall z. (\forall l.\ \mathsf{A}(l) = z \Rightarrow p(l) = \varnothing) \Rightarrow \\ \quad \exists r, M^r, M, M', ch.\ p[ch{:=}\mathsf{channel}], O, I, C, T \models \mathsf{wpcx}_n(\mathsf{val}(z), \xi) \land \\ \quad \mathsf{A}(ch) = z \land \mathsf{R}(ch){=}r \land \mathsf{M}^\mathsf{r}(ch){=}M^r \land \mathsf{M}(ch){=}M \land \mathsf{M}'(ch){=}M' \land \mathsf{S}(ch){=}b$$

▶ **Lemma 5** (Weakest Precondition of send).

$$\forall n, e_1, e_2, \xi, p, O, I, C, T.\quad p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{send}(e_1, e_2), \xi) \Rightarrow \\ \quad \exists p_1, p_2, C_1, C_2, T_1, T_2, T_e, ch, m.\ p = p_1 {\uplus} p_2 \land C = C_1 {\uplus} C_2 \land T = T_1 {\uplus} T_2 \land T_e = T_2([\![e_1]\!]) \\ \quad \land\ \mathsf{A}(ch) = [\![e_1]\!] \land m = [\![e_2]\!] \land (\mathsf{M}^\mathsf{r}(ch) = \{\!\} \lor 0 < T_e) \land \mathsf{levels}(\mathsf{M}'(ch)(m)) \subseteq \mathsf{M}^\mathsf{r}(ch) \land \\ \quad p_2(ch) = \mathsf{channel} \land \\ \quad p_1, \varnothing, \varnothing, C_1, T_1 \models \mathsf{M}(ch)(m) \land \\ \quad (\mathsf{M}^\mathsf{r}(ch) = \{\!\} \Rightarrow p_2, O{-}\{\!\mathsf{O}(ch)\!\}{-}\mathsf{M}'(ch)(m), I, C_2, T_2 \models \mathsf{wpcx}_n(\mathsf{tt}, \xi)) \land \\ \quad (\mathsf{M}^\mathsf{r}(ch) \neq \{\!\} \Rightarrow p_2, O{-}\{\!\mathsf{O}(ch)\!\}{-}\mathsf{M}'(ch)(m), I, C_2, T_2[[\![e_1]\!]{:=}T_e{-}1] \models \mathsf{wpcx}_n(\mathsf{tt}, \xi))$$

▶ **Lemma 6** (Weakest Precondition of receive).

$$\forall n, e, \xi, p, O, I, C, T.\quad p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{receive}(e), \xi) \Rightarrow \\ \quad \exists ch.\ p(ch) = \mathsf{channel} \land (S(ch) = \mathsf{true} \lor 0 < C([\![e]\!])) \land \mathsf{O}(ch) \prec O \land \mathsf{O}(ch) \prec^r I \land \\ \quad (\neg \mathsf{S}(ch) \lor (O = \{\!\} \land \forall o \in I.\ o = \mathsf{O}(ch))) \land \mathsf{A}(e) = [\![ch]\!] \land \\ \quad \forall m, p_1, C_1, T_1.\ p_1, \varnothing, \varnothing, C_1, T_1 \models \mathsf{M}(ch)(m) \Rightarrow \\ \qquad p {\uplus} p_1, O {\uplus} \mathsf{M}'(ch)(m), I{-}\{\!\mathsf{O}(ch)\!\}, C[[\![e]\!]{:=}C([\![e]\!]){-}1] {\uplus} C_1, T {\uplus} T_1 \models \mathsf{wpcx}_n(\mathsf{val}(m), \xi)$$

▶ **Lemma 7** (Weakest Precondition of fork).

$$\forall n, c, \xi, p, O, I, C, T.\quad p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{fork}(c), \xi) \Rightarrow \\ \exists p_1, p_2, O_1, O_2, I_1, I_2, C_1, C_2, T_1, T_2. \\ \quad p{=}p_1 {\uplus} p_2 \land O{=}O_1 {\uplus} O_2 \land I{=}I_1 {\uplus} I_2 \land C{=}C_1 {\uplus} C_2 \land T{=}T_1 {\uplus} T_2 \land \\ \quad p_1, O_1, I_1, C_1, T_1 \models \mathsf{wpcx}_n(\mathsf{tt}, \xi) \land \\ \quad p_2, O_2, I_2, C_2, T_2 \models \mathsf{wp}_{n-1}(c, \lambda\_.\mathsf{obs}(\{\!\}, \{\!\}))$$

NewChannel
$\{\mathsf{true}\}\ \mathsf{new\_channel}\ \{\lambda a.\ \mathsf{channel}((a, r, R, b), (M_{index}, M_{args}), M')\}$

Send
$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch) * \mathsf{M}(ch)(m) * (\mathsf{M}^{\mathsf{r}}(ch){=}\{\!\} \vee \mathsf{tranditt}(a)) \wedge$
$\mathsf{levels}(\mathsf{M}'(ch)(m)) \subseteq \mathsf{M}^{\mathsf{r}}(ch) \wedge \mathsf{A}(ch) = a\}\ \mathsf{send}(a, m)$
$\{\lambda\_.\ \mathsf{obs}(O{-}\{\!\{\mathsf{O}(ch)\}\!\}{-}\mathsf{M}'(ch)(m), I) * \mathsf{channel}(ch)\}$

Receive
$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch) * (\mathsf{S}(ch) \vee \mathsf{credit}(a)) \wedge \mathsf{O}(ch) \prec O \wedge \mathsf{O}(ch) \prec^r I \wedge$
$(\neg\mathsf{S}(ch) \vee (O{=}\{\!\} \wedge \forall o{\in}I.\ o{=}\mathsf{O}(ch))) \wedge \mathsf{A}(ch) = a\}\ \mathsf{receive}(a)$
$\{\lambda m.\ \mathsf{obs}(O{\uplus}\mathsf{M}'(ch)(m), I{-}\{\!\{\mathsf{O}(ch)\}\!\}) * \mathsf{channel}(ch) * \mathsf{M}(ch)(m)\}$

Credit
$\{\mathsf{obs}(O) * \mathsf{channel}(ch)\}\ \mathsf{nop}\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{\!\{\mathsf{O}(ch)\}\!\}) * \mathsf{channel}(ch) * \mathsf{credit}(\mathsf{A}(ch))\}$

Trandit
$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch)\}\ \mathsf{nop}\ \{\lambda\_.\ \mathsf{obs}(O, I{\uplus}\{\!\{\mathsf{O}(ch)\}\!\}) * \mathsf{channel}(ch) * \mathsf{trandit}(\mathsf{A}(ch))\}$

Trandits
$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch)\}\ \mathsf{nop}\ \{\lambda\_.\ \mathsf{obs}(O, I{\uplus}\{\!\{\mathsf{O}(ch)^{\infty}\}\!\}) * \mathsf{channel}(ch) * \mathsf{trandit}^{\infty}(\mathsf{A}(ch))\}$

Figure 31 The proof rules ensuring deadlock-freedom of importer channels, where ghost information is associated with channel addresses via the channel permissions.

C.4 Correctness of Commands

We define *correctness of commands*, as shown in Definition 8, ensuring that each proposed proof rule, where $\mathsf{correct}_{pt}(P, c, Q)$ is abbreviated as $\{P\}\ c\ \{Q\}$, respects the definition of the weakest precondition. Having this definition we prove the proposed proof rules, ensuring deadlock freedom of importer channels, as well as some other necessary proof rules shown in Theorems 9, 10, and 11.

▶ **Definition 8** (Correctness of Commands). *A command is correct w.r.t a precondition P and a postcondition Q if and only if P implies the weakest precondition of that command w.r.t Q.*

$$\mathsf{correct}_{pt}(P, c, Q) \Leftrightarrow \forall n.\ P \Rightarrow \mathsf{wp}_{n,pt}(c, Q)$$

▶ **Theorem 9** (Rule Sequential Composition).

$$\mathsf{correct}(P, c_1, Q) \wedge (\forall z.\ \mathsf{correct}(Q(z), c_2[z/x], R)) \Rightarrow \mathsf{correct}(P, \mathsf{let}(x, c_1, c_2), R)$$

▶ **Theorem 10** (Rule Consequence).

$$\mathsf{correct}(P, c, Q) \wedge (P' \vdash P) \wedge (\forall z.\ Q(z) \vdash Q'(z)) \Rightarrow \mathsf{correct}(P', c, Q')$$

▶ **Theorem 11** (Rule Frame).

$$\mathsf{correct}(P, c, Q) \Rightarrow \mathsf{correct}(P * F, c, \lambda z.\ Q(z) * F)$$

As previously mentioned, since in this formalization ghost information is associated with channel addresses via the channel permissions rather than via global functions, we provide a new version of the proof rules, proposed in Section 2, associated with this formalization as shown in Figure 31.

C.5 Validity of a Configuration

We define *validity of a configuration*, shown in Definition 12, and prove that 1) starting from a valid configuration, all the subsequent configurations of the execution are also valid (Theorem 13), 2) a valid configuration is not deadlocked (Theorem 14), and 3) if a program c is verified by the proposed proof rules, where the verification starts from empty bags of obligations and importers and ends with such bags too, then the initial configuration, where the heap is empty, denoted by $\mathbf{0}{=}\lambda\_.\varnothing$, and there is only one thread with the command c (and a context done), and the list of server channels is empty, is a valid configuration (Theorem 16).

▶ **Definition 12** (Validity of a Configuration). *A configuration (t, h, s) is valid for n steps, denoted by $\mathsf{valid}_n(t, h, s)$, if there exists a set of* augmented threads *A, consisting of the identifier (id), the program (c), the context (ξ), the permission heap (p), the obligations (O), the importers (I), the credits (C), and the trandits (T) associated with each thread such that all of the following conditions hold:*

1. $\forall id, c, \xi.\ t(id) = (c; \xi) \Leftrightarrow \exists p, O, I, C, T.\ (id, c, \xi, p, O, I, C, T) \in A$
2. $\forall (id_1, c_1, \xi_1, p_1, O_1, I_1, C_1, T_1) \in A, (id_2, c_2, \xi_2, p_2, O_2, I_2, C_2, T_2) \in A.\ id_1 = id_2 \Rightarrow$ $(id_1, c_1, \xi_1, p_1, O_1, I_1, C_1, T_1) = (id_2, c_2, \xi_2, p_2, O_2, I_2, C_2, T_2)$
3. $\forall l_1, l_2.\ \mathsf{Pt}(l_1) \neq \varnothing \wedge \mathsf{Pt}(l_2) \neq \varnothing \wedge \mathsf{A}(l_1) = \mathsf{A}(l_1) \Rightarrow l_1 = l_2$
4. $\forall l.\ \mathsf{Pt}(l) \neq \varnothing \Rightarrow h(\mathsf{A}(l)) \neq \varnothing$ *and* $\forall z.\ (\forall l.\ \mathsf{A}(l) = z \Rightarrow \mathsf{Pt}(l) = \varnothing) \Rightarrow h(z) = \varnothing$
5. $\forall ch.\ \mathsf{Pt}(ch){=}\mathsf{channel} \Rightarrow$
   a. $\mathsf{M}^{\mathsf{r}}(ch) \neq \{\!\}\!\} \Rightarrow \mathsf{Tt}(\mathsf{A}(ch)) + \mathsf{size}_h(ch) \leqslant \mathsf{It}(ch) \wedge$
   b. $\forall m \in \mathsf{queue}_h(ch).\ \mathsf{levels}(\mathsf{M}'(ch)(m)) \subseteq \mathsf{M}^{\mathsf{r}}(ch) \wedge$
   c. $\mathsf{S}(ch) = \mathsf{false} \Rightarrow$
   $\mathsf{Ct}(\mathsf{A}(ch)) \leqslant \mathsf{Ot}(ch) + \mathsf{size}_h(ch) + \sum_{ch'\ where\ \mathsf{Pt}(ch')=\mathsf{channel}} \sum_{m \in \mathsf{queue}_h(ch')} \mathsf{M}'(ch')(m)(ch)$
6. $\forall (id, c, \xi, p, O, I, C, T) \in A.\ p, O, I, C, T \models \mathsf{wpcx}_n(c, \xi)$
7. $\forall z \in s.\ \exists ch.\ \mathsf{Pt}(ch) = \mathsf{channel} \wedge \mathsf{S}(ch) = \mathsf{true} \wedge \mathsf{A}(ch) = z$
8. $\forall ch.\ \mathsf{Pt}(ch) = \mathsf{channel} \wedge \mathsf{S}(ch) = \mathsf{true} \Rightarrow \mathsf{A}(ch) \in s$

where

- $\mathsf{size}_h(ch)$ *returns the number of the messages in the channel ch, i.e.* $|h(\mathsf{A}(ch))|$
- $\mathsf{queue}_h(ch)$ *returns the messages in the channel ch, i.e.* $h(\mathsf{A}(ch))$
- $\mathsf{levels}(O)$ *returns the levels of the obligations in O, i.e.* $\{\!\{r \mid (a, r) \in O\}\!\}$
- $\mathsf{Pt} = \biguplus_{(id,c,\xi,p,O,I,C,T)\in A} p$ *and* $\mathsf{Wt} = \biguplus_{(id,\mathsf{wait}(l),\xi,p,O,I,C,T)\in A} \{\!\{l\}\!\}$
- $\mathsf{Ct} = \biguplus_{(id,c,\xi,p,O,I,C,T)\in A} C$ *and* $\mathsf{Ot} = \biguplus_{(id,c,\xi,p,O,I,C,T)\in A} O$
- $\mathsf{Tt} = \biguplus_{(id,c,\xi,p,O,I,C,T)\in A} T$ *and* $\mathsf{It} = \biguplus_{(id,c,\xi,p,O,I,C,T)\in A} I$

▶ **Theorem 13** (Small Steps Preserve Validity of Configurations). *Each step of the execution preserves validity of configurations.*

$$(t, h, s) \rightsquigarrow (t', h', s') \wedge \mathsf{valid}_{n+1}(t, h, s) \Rightarrow \mathsf{valid}_n(t', h', s')$$

Proof. By case analysis of the small step relation $\rightsquigarrow$.
Case $(t[id{:=}\mathsf{new\_channel}(b); \xi], h[z{:=}\varnothing], s) \rightsquigarrow (t[id{:=}\mathsf{val}(z); \xi], h[z{:=}[]], b{=}\mathsf{true}\ ?\ s{\cup}\{z\} : s)$:
By $\mathsf{valid}(t[id{:=}\mathsf{new\_channel}(b); \xi], h[z{:=}\varnothing], s)$ we have an augmented thread set A consisting of an element $(id, \mathsf{new\_channel}(b), \xi, p, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations, including $p, O, I, C, T \models \mathsf{wpcx}(\mathsf{new\_channel}(b), \xi)$. $\mathsf{valid}(t[id{:=}\mathsf{val}(z); \xi], h[ch{:=}[]], b{=}\mathsf{true}\ ?\ s{\cup}\{z\} : s)$ holds because by Lemma 4 there exists an augmented thread set $A'{=}A{-}(id, \mathsf{new\_channel}(b), \xi, p, O, I, C, T){\cup}(id, \mathsf{val}(z), \xi, p[l{:=}\mathsf{channel}]$

$, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations.

Case $(t[id{:=}\mathsf{send}(e_1, e_2); \xi], h[[\![ch]\!]{:=}M], s) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h[[\![e_1]\!]{:=}M.[[\![e_2]\!]]], s)$:
By $\mathsf{valid}(t[id{:=}\mathsf{send}(e_1, e_2); \xi], h[[\![ch]\!]{:=}M], s)$ we have an augmented thread set A consisting of an element $(id, \mathsf{send}(e_1, e_2), \xi, p, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations, including $p, O, I, C, T \models \mathsf{wpcx}(\mathsf{send}(e_1, e_2), \xi)$. $\mathsf{valid}(t[id{:=}\mathsf{tt}; \xi], h[[\![e_1]\!]{:=}M.[[\![e_2]\!]]], s)$ holds because by Lemma 5 there exists an augmented thread set $A'{=}A{-}(id, \mathsf{send}(e_1, e_2), \xi, p, O', I', C, T) \cup (id, \mathsf{tt}, \xi, p_2, O \uplus \mathsf{M}'(ch)(m), I{-}\{\!\{ch\}\!\}, C_2, T_2)$ which satisfies all the conditions in the definition of validity of configurations.

Case $(t[id{:=}\mathsf{receive}(e); \xi], h[[\![e]\!]{:=}[m].M], s) \rightsquigarrow (t[id{:=}\mathsf{val}(m); \xi], h[[\![e]\!]{:=}M], s)$:
By $\mathsf{valid}(t[id{:=}\mathsf{receive}(e); \xi], h[[\![e]\!]{:=}[m].M], s)$ we have an augmented thread set A consisting of an element $(id, \mathsf{receive}(e), \xi, p, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations, including $p, O, I, C, T \models \mathsf{wpcx}(\mathsf{receive}(e), \xi)$. $\mathsf{valid}(t[id{:=}\mathsf{val}(m); \xi], h[[\![e]\!]{:=}M], s)$ holds because by Lemma 6 there exists an augmented thread set $A'{=}A{-}(id, \mathsf{receive}(e), \xi, p, O, I, C, T) \cup (id, \mathsf{val}(m), \xi, p_2, O \uplus \mathsf{M}'(ch)(m), I{-}\{\!\{ch\}\!\}, C_2, T_2)$ which satisfies all the conditions in the definition of validity of configurations.

Case $(t[id{:=}\mathsf{fork}(c); \xi, id'{:=}\varnothing], h, s) \rightsquigarrow (t[id{:=}\mathsf{val}(\mathsf{tt}); \xi, id'{:=}c; \mathsf{done}], h, s)$:
By $\mathsf{valid}_n(t[id{:=}\mathsf{fork}(c); \xi, id'{:=}\varnothing]; \xi, h, s)$ we have an augmented thread set A consisting of an element $(id, \mathsf{fork}(c), \xi, p, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations, including $p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{fork}(c), \xi)$. $\mathsf{valid}(t[id{:=}\mathsf{val}(\mathsf{tt}); \xi, id'{:=}c; \mathsf{done}], h, s)$ holds because by Lemma 7 there exists an augmented thread set $A'{=}A{-}(id, \mathsf{fork}(c), \xi, p, O, I, C, T) \cup (id, \mathsf{tt}, \xi, p_1, O_1, I_1, C_1, T_1) \cup (id', c, \mathsf{done}, p_2, O_2, I_2, C_2, T_2)$ which satisfies all the conditions in the definition of validity of configurations.

Case $(t[id{:=}\mathsf{let}(x, c_1, c_2); \xi], h, s) \rightsquigarrow (t[id{:=}c1; \mathsf{let}'(x, c_2, \xi)], h, s)$:
By $\mathsf{valid}_n(t[id{:=}\mathsf{let}(x, c_1, c_2); \xi], h)$ we have an augmented thread set A consisting of an element $(id, \mathsf{let}(x, c_1, c_2), \xi, p, O, g)$ which satisfies all the conditions in the definition of validity of configurations, including $p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{let}(x, c_1, c_2), \xi)$. Since $\mathsf{wpcx}_n(\mathsf{let}(x, c_1, c_2), \xi) = \mathsf{wp}_{n-1}(c_1, \lambda z.\ \mathsf{wp}_{n-1}(c_2[z/x], Q))$, we have $p, O, I, C, T \models \mathsf{wp}_{n-1}(c_1, \lambda z.\ \mathsf{wp}_{n-1}(c_2[z/x], Q))$. Consequently $\mathsf{valid}(t[id{:=}c1; \mathsf{let}'(x, c_2, \xi)], h, s)$ holds because there exists an augmented thread set $A'{=}A{-}(id, \mathsf{let}(x, c_1, c_2), \xi, p, O, I, C, T) \cup (id, c_1, \mathsf{let}'(x, c_2, \xi), p, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations. The rest of the cases can be proved similarly. ◀

▶ **Theorem 14** (A Valid Configuration Is Not Deadlocked). *If a valid configuration has some threads then either all threads in this configuration are waiting for some server channels, or there exists a thread in this configuration which is not waiting for an empty channel.*

$$\mathsf{valid}_n(t, h, s) \wedge \exists id.\ t(id) \neq \varnothing \Rightarrow \mathsf{NotDeadlock}(t, h, s)$$

where $\mathsf{NotDeadlock}(t, h, s) \Leftrightarrow \mathsf{AllWaitingToServe}(t, s) \vee \exists id'.\ \neg\mathsf{is\_waiting}(\mathsf{fst}(t(id')), h)$, *where*

- $\mathsf{AllWaitingToServe}(t, s) \Leftrightarrow \forall id.\ t(id) \neq \varnothing \Rightarrow \exists e.\ \mathsf{fst}(t(id)) = \mathsf{wait}(e) \wedge [\![e]\!] \in s$
- $\mathsf{is\_waiting}(c, h) \Leftrightarrow \exists e.\ c = \mathsf{wait}(e) \wedge h([\![e]\!]) = []$.

Proof. By contradiction; we assume that all threads in t are waiting for some empty channels where some of these channels are not server channels, i.e. $\forall id.\ \exists e.\ \mathsf{fst}(t(id)){=}\mathsf{wait}(e) \wedge h(e){=}[] \wedge \exists id', e'.\ \mathsf{fst}(t(id')){=}\mathsf{wait}(e') \wedge [\![e']\!] \notin s$. Since (t, h, s) is a valid configuration and all threads in this configuration are waiting for a channel, there exists a set of valid augmented

threads A from which we produce a valid bag $G = \mathsf{valid\_bag}(A)$, where $\mathsf{valid\_bag}$ maps any element $(id, \mathsf{wait}(e), \xi, p, O, I, C, T) \in A$ to an element $([\![e]\!], \mathsf{Addresses}(O), \mathsf{Addresses}(I))$ where $\mathsf{Addresses}(O) = \{\!|\mathsf{A}(o) \mid o \in O|\!\}$. By Lemma 15, we have $G = \{\!|\}\!\}$, implying $A = \{\}$, implying $t = \mathbf{0}$ which contradicts the hypothesis of the theorem.

Note that in the definition of validity of a configuration we also keep track of all locations whose addresses are allocated, which makes it possible to provide the functions R, M^r, and S, mapping channel addresses to their ghost information, for Lemma 15. Additionally, the hypotheses H_2, H_3, H_4, and $H_5 \vee H_6$ in Lemma 15 are met as follows. For each element $(id, \mathsf{wait}(e), \xi, p, O, I, C, T) \in A$ we have $p, O, I, C, T \models \mathsf{wpcx}(\mathsf{wait}(e), \xi)$, which implies $0 < C([\![e]\!])$ and there exists a channel ch with address $[\![e]\!]$ such that $ch \prec O$ (which implies H_2) and $ch \prec^r I$ (which implies H_3), and $\mathsf{S}(ch) = \mathsf{true} \Rightarrow O = \{\!|\}\!\} \wedge \forall o_1.0 < I(o_1) \Rightarrow o_1 = ch$ (which implies H_4). Additionally, by $0 < C([\![e]\!])$ we have $0 < \mathsf{Ct}([\![e]\!])$, which (by $4.c$ in validity of configurations) implies if $\mathsf{S}(ch) = \mathsf{false}$ either 1) $0 < \mathsf{Ot}(ch)$ (which implies H_5), or 2) there exists a message m in a channel ch' through which an obligation of ch is transferred, i.e. $0 < \mathsf{size}(ch')$ and $m \in \mathsf{queue}_h(ch')$ and $0 < \mathsf{M'}(ch')(m)(ch)$. By $m \in \mathsf{queue}_h(ch')$ and $4.b$ we have $\mathsf{levels}(\mathsf{M'}(ch')(m)) \subseteq \mathsf{M^r}(ch')$, which by $\{\!|ch|\!\} \in \mathsf{M'}(ch')(m)$ implies $\mathsf{R}(ch) \in \mathsf{M^r}(ch')$. Additionally, by $\mathsf{M^r}(ch') \neq \{\!|\}\!\}$, and $0 < \mathsf{size}(ch')$, and $4.a$ we have $0 < \mathsf{It}(ch')$ (which implies H_6). ◀

Lemma 15 ensures that in any state of the execution if all the desired invariants are respected then it is impossible that all threads of the program are waiting for some empty channels where some of these channels are not server channels. In this lemma G is a bag of waitable object-obligations-importers triples 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 associated with the address of the object for which t is waiting, the second element is associated with the addresses of obligations of t, and the third element is associated with the addresses of importers of t.

▶ **Lemma 15** (A Valid Bag of Augmented Threads Is Not Deadlocked)**.**

$$\begin{aligned}&\forall\, G : \mathit{Bags}(\mathit{Addresses} \times \mathit{Bags}(\mathit{Addresses}) \times \mathit{Bags}(\mathit{Obligations})),\\ &\quad R : \mathit{Addresses} \to \mathit{Levels},\\ &\quad M^r : \mathit{Addresses} \to \mathit{Bags}(\mathit{Levels}),\\ &\quad S : \mathit{Addresses} \to \mathit{Booleans}.\\ &H_1 \wedge \forall (o, O, I) \in G.\ H_2 \wedge H_3 \wedge H_4 \wedge (H_5 \vee H_6) \Rightarrow G = \{\!|\}\!\}\end{aligned}$$

where

- $H_1 : \exists (o, O, I) \in G.\ S(o) = \mathsf{false}$
- $H_2 : o \prec O$
- $H_3 : o \prec^r I$
- $H_4 : S(o) = \mathsf{true} \Rightarrow O = \{\!|\}\!\} \wedge \forall o_1.\ 0 < I(o_1) \Rightarrow o_1 = o$
- $H_5 : S(o) = \mathsf{false} \Rightarrow 0 < \mathsf{Ot}(o)$
- $H_6 : S(o) = \mathsf{false} \Rightarrow \exists o_1.\ R(o) \in \mathsf{M^r}(o_1) \wedge \mathsf{Wt}(o_1) = 0 \wedge 0 < \mathsf{It}(o_1)$

  where $\mathsf{Wt} = \biguplus_{(o,O,I) \in G} \{\!|o|\!\}$ *and* $\mathsf{Ot} = \biguplus_{(o,O,I) \in G} O$ *and* $\mathsf{It} = \biguplus_{(o,O,I) \in G} I$

Proof. By H_1 we know $\exists (o_m, O_1, I_1) \in G$ where $S(o_m) = \mathsf{false}$ and $\forall (o, O, I) \in G.\ S(o) = \mathsf{false} \Rightarrow R(o_m) \leqslant R(o)$. By $(H_5 \vee H_6)$ there are two cases: 1) $\exists (o_2, \{\!|o_m|\!\} \uplus O_2, I_2) \in G$, or 2) $\exists (o_3, O_3, \{\!|o_1|\!\} \uplus I_3) \in G.\ R(o_m) \in M^r(o_1) \wedge \mathsf{Wt}(o_1) = 0$. In the first case by H_4 we have $S(o_2) = \mathsf{false}$, which implies $R(o_m) \leqslant R(o_2)$, that contradicts the hypothesis H_2, i.e $o_2 \prec \{\!|o_m|\!\} \uplus O_2$. In the second case by $\mathsf{Wt}(o_1) = 0$ we have $o_1 \neq o_3$ (because $0 < \mathsf{Wt}(o_3)$), and consequently by H_4 we have $S(o_3) = \mathsf{false}$, which implies $R(o_m) \leqslant R(o_3)$, that contradicts the hypothesis H_3, i.e. $o_3 \prec^r \{\!|o_1|\!\} \uplus I_3$ (because $R(o_m) \in M^r(o_1)$). ◀

▶ **Theorem 16** (The Initial Configuration Is Valid)**.**

$$\mathsf{correct}(\mathsf{obs}(\{\},\{\}), c, \lambda\_.\mathsf{obs}(\{\},\{\})) \Rightarrow \forall n, id.\ \mathsf{valid}_n(\mathbf{0}[id{:=}c; \mathsf{done}], \mathbf{0}, [])$$

Proof. The goal is achieved because there exists an augmented thread set $A{=}[(id, c, \mathsf{done}, \mathbf{0}, \mathit{0}, \mathit{0}, \mathit{0}, \mathit{0})]$, such that all the conditions in the definition of validity of configurations are met, where $\mathbf{0} = \lambda\_.\varnothing$ and $\mathit{0} = \lambda\_.0$. ◀

C.6 An Example Proof

In this section we show how the program in Figure 3 can be verified using the proof rules in Figure 31, as shown in Figure 32.

D Transferring Obligations Through Notifications: Soundness Proof

In this appendix we provide a formalization and soundness proof, machine-checked with Coq[11], for the approach introduced in Section 3. However, unfortunately, there are a few technical differences between this formalization and the system of Section 3 such that in this formalization the ghost information, such as level and transferred permissions and obligations, are associated with lock and condition variable addresses via the lock and cond permissions rather than via global functions[12]. The proof rules associated with this formalization and the verification of the program in Figure 16, proved using these rules, are shown in Sections D.4 and D.6, respectively.

D.1 Syntax and Semantics of Programs

We define the syntax of our programming language as indicated in Figure 33. In this syntax an arithmetic expression, e, can be an integer value, z, a variable, x, an addition of two expressions, or a negation of an expression. An integer value can be substituted for a free variable in an expression or a command, and each expression can be evaluated to an integer value, as shown in Figure 34. Commands include commands $\mathsf{val}(e)$ which simply yield the value of e as their result and have no side effects, memory allocations[13], memory reads, memory writes, conditionals, loops, parallel composition, sequential composition, lock creations, lock acquisitions, lock releases, condition variable creations, waits, and notifications. We also define some extra commands waiting4lock, indicating that the related thread is waiting for a lock, and waiting4cvar, indicating that the related thread has executed wait; these are not supposed to appear in the source program and appear only during execution. Additionally, instead of defining all the ghost commands introduced in Section 3, we define a single ghost command nop which is inserted into the program for verification purposes and has no effect on the program's behavior. The small step semantics, defined in Figure 36, relates two *configurations*, defined in Figure 35.

$\mathsf{ob}(a) ::= (a, 1, \{\!|1|\!\}, \mathsf{false})$
$\mathsf{ob}'(a) ::= (a, 1, \{\!|\}\!\}, \mathsf{false})$
$\mathsf{loc}(a) ::= (\mathsf{ob}(a), (\mathsf{Mch}, []), \lambda m.\ \{\!|\mathsf{ob}'(\mathsf{snd}(m))|\!\})$
$\mathsf{loc}'(a) ::= (\mathsf{ob}'(a), (\mathsf{Mch}', []), \lambda\_.\ \{\!|\}\!\})$
$\mathsf{pt}(\mathsf{Mch}, args) ::= \lambda m.\ \mathsf{channel}(\mathsf{loc}'(\mathsf{snd}(m)))$
$\mathsf{pt}(\mathsf{Mch}', args) ::= \lambda m.\ \mathsf{true}$

routine server(channel a){
req : $\{\mathsf{obs}(\{\!|\}\!\}, \{\!|\mathsf{ob}(a)|\!\}) * \mathsf{channel}(\mathsf{loc}(a)) * \mathsf{credit}(a)\}$
$(req, a') := \mathsf{receive}(ch)$;
$\{\mathsf{obs}(\{\!|\mathsf{ob}'(a')|\!\}, \{\!|\}\!\}) * \mathsf{channel}(\mathsf{loc}'(a'))\}$
$result := \mathsf{process}(req)$;
$\mathsf{send}(a', result)$
ens : $\{\mathsf{obs}(\{\!|\}\!\}, \{\!|\}\!\})\}$}

routine client(channel a){
req : $\{\mathsf{obs}(\{\!|\mathsf{ob}(a)|\!\}, \{\!|\}\!\}) * \mathsf{channel}(\mathsf{loc}(a)) * \mathsf{trandit}(a)\}$
$a' := \mathsf{new\_channel}()$;
$\{\mathsf{obs}(\{\!|\mathsf{ob}(a)|\!\}, \{\!|\}\!\}) * \mathsf{channel}(\mathsf{loc}'(a')) * \mathsf{trandit}(a)\}$
nop; //Rule CREDIT
$\{\mathsf{obs}(\{\!|\mathsf{ob}(a), \mathsf{ob}'(a')|\!\}, \{\!|\}\!\}) * \mathsf{trandit}(a) * \mathsf{credit}(a')\}$
$\mathsf{send}(a, (\mathsf{request}(), a'))$;
$\{\mathsf{obs}(\{\!|\}\!\}, \{\!|\}\!\}) * \mathsf{credit}(a')\}$
$\mathsf{receive}(a')$
ens : $\{\mathsf{obs}(\{\!|\}\!\}, \{\!|\}\!\})\}$}

routine main(){
req : $\{\mathsf{obs}(\{\!|\}\!\}, \{\!|\}\!\})\}$
$a := \mathsf{new\_channel}()$;
$\{\mathsf{obs}(\{\!|\}\!\}, \{\!|\}\!\}) * \mathsf{channel}(\mathsf{loc}(a))\}$
nop; //Rule CREDIT
$\{\mathsf{obs}(\{\!|\mathsf{ob}(a)|\!\}, \{\!|\}\!\}) * \mathsf{credit}(a)\}$
nop; //Rule TRANDIT
$\{\mathsf{obs}(\{\!|\mathsf{ob}(a)|\!\}, \{\!|\mathsf{ob}(a)|\!\}) * \mathsf{credit}(a) * \mathsf{trandit}(a)\}$
fork(
  $\{\mathsf{obs}(\{\!|\}\!\}, \{\!|\mathsf{ob}(a)|\!\}) * \mathsf{credit}(a)\}$
  $\mathsf{server}(a)$
  $\{\mathsf{obs}(\{\!|\}\!\}, \{\!|\}\!\})\}$);
$\{\mathsf{obs}(\{\!|\mathsf{ob}(a)|\!\}, \{\!|\}\!\}) * \mathsf{trandit}(a)\}$
$\mathsf{client}(a)$
ens : $\{\mathsf{obs}(\{\!|\}\!\}, \{\!|\}\!\})\}$}

Figure 32 Verification of the program in Figure 3 using the rules in Figure 31.

$c \in \mathit{Commands},\ e \in \mathit{Expressions},\ z \in \mathbb{Z},\ x \in \mathit{Variables}$
$e ::= z \mid x \mid e_1{+}e_2 \mid -e$
$c ::= \mathsf{val}(e) \mid \mathsf{new\_int}(z) \mid \mathsf{lookup}(e) \mid \mathsf{mutate}(e_1, e_2)$
$\quad \mid \mathsf{if}(c, c_1, c_2) \mid \mathsf{while}(c, c_1) \mid \mathsf{let}(x, c_1, c_2) \mid \mathsf{fork}(c) \mid \mathsf{new\_lock} \mid \mathsf{acquire}(e) \mid \mathsf{release}(e)$
$\quad \mid \mathsf{new\_cvar} \mid \mathsf{wait}(e_1, e_2) \mid \mathsf{notify}(e) \mid \mathsf{notifyAll}(e)$
$\quad \mid \mathsf{waiting4lock}(e) \mid \mathsf{waiting4cvar}(e_1, e_2) \mid \mathsf{nop}$

Figure 33 Syntax of the programming language

$[\![]\!] \in \mathit{Expressions} \to \mathbb{Z}$

$[\![z]\!]{=}z$
$[\![x]\!]{=}0$
$[\![e_1{+}e_2]\!]{=}[\![e_1]\!]{+}[\![e_2]\!]$
$[\![-e]\!]{=}-[\![e]\!]$

$z[z'/x] = z$
$x[z/x'] = x \quad \text{if } x \neq x'$
$x[z/x] = z$
$(e_1{+}e_2)[z/x] = e_1[z/x]{+}e_2[z/x]$

$\mathsf{val}(e)[z/x] = \mathsf{val}(e[z/x])$
$\mathsf{let}(x, c_1, c_2)[z/x] = \mathsf{let}(x, c_1, c_2)$
$\mathsf{let}(x, c_1, c_2)[z/x'] = \mathsf{let}(x, c_1[z/x'], c_2[z/x']) \quad \text{if } x \neq x'$
$\mathsf{fork}(c)[z/x] = \mathsf{fork}(c[z/x])$
$\mathsf{new\_lock}[z/x] = \mathsf{new\_lock}$
$\mathsf{new\_int}(z')[z/x] = \mathsf{new\_int}(z')$
$\mathsf{lookup}(e)[z/x] = \mathsf{lookup}(e[z/x])$
$\mathsf{mutate}(e_1, e_2)[z/x] = \mathsf{mutate}(e_1[z/x], e_2[z/x])$
$\mathsf{acquire}(e)[z/x] = \mathsf{acquire}(e[z/x])$
$\mathsf{release}(e)[z/x] = \mathsf{release}(e[z/x])$
$\mathsf{new\_cvar}[z/x] = \mathsf{new\_cvar}$
$\mathsf{wait}(e_1, e_2)[z/x] = \mathsf{wait}(e_1[z/x], e_2[z/x])$
$\mathsf{notify}(e)[z/x] = \mathsf{notify}(e[z/x])$
$\mathsf{notifyAll}(e)[z/x] = \mathsf{notifyAll}(e[z/x])$
$\mathsf{waiting4lock}(e)[z/x] = \mathsf{waiting4lock}(e[z/x])$
$\mathsf{waiting4cvar}(e_1, e_2)[z/x] = \mathsf{waiting4cvar}(e_1[z/x], e_2[z/x])$
$\mathsf{while}(c, c_1)[z/x] = \mathsf{while}(c[z/x], c_1[z/x])$
$\mathsf{if}(c, c_1, c_2)[z/x] = \mathsf{if}(c[z/x], c_1[z/x], c_2[z/x])$
$\mathsf{nop}[z/x] = \mathsf{nop}$

Figure 34 Evaluation of expressions and substitution of expressions and commands

$$
\begin{array}{l}
\mathit{Addresses} = \mathbb{Z} \\
\mathit{ThreadIds} = \mathbb{Z} \\
h \in \mathit{Heaps} = \mathit{Addresses} \rightharpoonup \mathbb{Z} \\
\xi \in \mathit{Contexts} ::= \mathsf{done} \mid \mathsf{let}'(x, c, \xi) \mid \mathsf{if}'(c_1, c_2, \xi) \\
\theta \in \mathit{ThreadConfigurations} ::= (c; \xi) \\
t \in \mathit{ThreadTables} = \mathit{ThreadIds} \rightharpoonup \mathit{ThreadConfigurations} \\
\kappa \in \mathit{Configurations} = \mathit{Heaps} \times \mathit{ThreadTables}
\end{array}
$$

Figure 35 Configurations

$$
\begin{array}{l}
(t[id{:=}\mathsf{new\_int}(n); \xi], h[z...z{+}n{-}1{:=}\varnothing]) \rightsquigarrow (t[id{:=}\mathsf{val}(z); \xi], h[z...z{+}n{-}1{:=}0]) \\
(t[id{:=}\mathsf{lookup}(e); \xi], h[[\![e]\!]{:=}z]) \rightsquigarrow (t[id{:=}\mathsf{val}(z); \xi], h[[\![e]\!]{:=}z]) \\
(t[id{:=}\mathsf{mutate}(e_1, e_2); \xi], h) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h[[\![e_1]\!]{:=}[\![e_2]\!]]) \\
(t[id{:=}\mathsf{if}(c, c_1, c_2); \xi], h) \rightsquigarrow (t[id{:=}c; \mathsf{if}'(c1, c2, \xi)], h) \\
(t[id{:=}\mathsf{val}(e); \mathsf{if}'(c_1, c_2, \xi)], h) \rightsquigarrow (t[id{:=}c_1; \xi], h) \quad \text{if } 0{<}[\![e]\!] \\
(t[id{:=}\mathsf{val}(e); \mathsf{if}'(c_1, c_2, \xi)], h) \rightsquigarrow (t[id{:=}c_2; \xi], h) \quad \text{if } [\![e]\!]{\leqslant}0 \\
(t[id{:=}\mathsf{while}(c, c_1); \xi], h) \rightsquigarrow (t[id{:=}\mathsf{if}(c, \mathsf{let}(x, c_1, \mathsf{while}(c, c_1)), \mathsf{tt}), \xi], h) \\
\quad \text{where } x \text{ is not free in } c \text{ and } c_1 \\
(t[id{:=}\mathsf{fork}(c); \xi, id'{:=}\varnothing], h) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi, id'{:=}c; \mathsf{done}], h) \\
(t[id{:=}\mathsf{let}(x, c_1, c_2); \xi], h) \rightsquigarrow (t[id{:=}c_1; \mathsf{let}'(x, c_2, \xi)], h) \\
(t[id{:=}\mathsf{val}(e); \mathsf{let}'(x, c, \xi)], h) \rightsquigarrow (t[id{:=}c[[\![e]\!]/x]; \xi], h) \\
(t[id{:=}\mathsf{val}(e); \mathsf{done}], h) \rightsquigarrow (t[id{:=}\varnothing], h) \\
(t[id{:=}\mathsf{new\_lock}; \xi], h[z{:=}\varnothing]) \rightsquigarrow (t[id{:=}\mathsf{val}(z); \xi], h[z{:=}1]) \\
(t[id{:=}\mathsf{acquire}(e); \xi], h[[\![e]\!]{:=}1]) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h[[\![e]\!]{:=}0]) \\
(t[id{:=}\mathsf{acquire}(e); \xi], h[[\![e]\!]{:=}0]) \rightsquigarrow (t[id{:=}\mathsf{waiting4lock}(e); \xi], h[[\![e]\!]{:=}0]) \\
(t[id{:=}\mathsf{waiting4lock}(e); \xi], h[[\![e]\!]{:=}1]) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h[[\![e]\!]{:=}0]) \\
(t[id{:=}\mathsf{release}(e); \xi], h) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h[[\![e]\!]{:=}1]) \\
(t[id{:=}\mathsf{new\_cvar}; \xi], h[z{:=}\varnothing]) \rightsquigarrow (t[id{:=}\mathsf{val}(z); \xi], h[z{:=}0]) \\
(t[id{:=}\mathsf{wait}(e_1, e_2); \xi], h) \rightsquigarrow (t[id{:=}\mathsf{waiting4cvar}(e_1, e_2); \xi], h[[\![e_2]\!]{:=}1]) \\
(t[id{:=}\mathsf{notify}(e); \xi, id'{:=}\mathsf{waiting4cvar}(e_1, e_2); \xi'], h) \rightsquigarrow \\
\quad (t[id{:=}\mathsf{val}(\mathsf{tt}); \xi, id'{:=}\mathsf{waiting4lock}(e_2); \xi'], h) \quad \text{if } [\![e]\!] = [\![e_1]\!] \\
(t[id{:=}\mathsf{notify}(e); \xi], h) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h) \qquad \text{if } \mathsf{nowaiting}([\![e]\!], t) \\
(t[id{:=}\mathsf{notifyAll}(e); \xi], h) \rightsquigarrow (\mathsf{wkup}([\![e]\!], t[id{:=}\mathsf{val}(\mathsf{tt}); \xi]), h) \\
(t[id{:=}\mathsf{nop}; \xi], h) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h) \\
\text{where} \\
\mathsf{wkup}(z, t) = \lambda id.\ \begin{cases} \mathsf{waiting4lock}(l); \xi & \text{if } t(id) = \mathsf{waiting4cvar}(v, l); \xi \wedge [\![v]\!] = z \\ t(id) & \text{otherwise} \end{cases} \\
\\
\mathsf{nowaiting}(z, t) \Leftrightarrow \nexists id, \xi, l, v.\ [\![v]\!]{=}z \wedge t(id){=}\mathsf{waiting4cvar}(v, l); \xi
\end{array}
$$

Figure 36 Semantics of programs, where tt stands for val(0).

$$
\begin{array}{l}
\mathit{Bags}(A) = A \to \mathbb{N} \\
\mathit{Wt}, \mathit{Ot} \in \mathit{Bags}(\mathbb{Z}) \\
\mathit{Indexes} = \mathbb{Z} \\
\mathit{Arguments} = \mathbb{Z} \\
r \in \mathit{Levels} = \mathbb{R} \\
o \in \mathit{Obligations} = \mathit{Addresses} \times \mathit{Levels} \times \mathit{Addresses} \\
l \in \mathit{Locations} = \mathit{Obligations} \times (\mathit{Indexes} \times \mathit{Lists}(\mathit{Arguments})) \times \\
\quad (\mathit{Indexes} \times \mathit{Lists}(\mathit{Arguments})) \times \mathit{Bags}(\mathit{Obligations}) \\
O \in \mathit{Bags}(\mathit{Obligations}) \\
b \in \mathit{Booleans} \\
\hat{v} \in \mathit{AValues} ::= z \mid r \mid b \mid l \mid o \mid O \\
\alpha \in \mathit{AValues} \to \mathit{Assertions} \\
\pi \in \mathit{Fractions} \\
a \in \mathit{Assertions} ::= l \overset{\pi}{\mapsto} z \mid \mathsf{ulock}(l, \mathit{Wt}, \mathit{Ot}) \mid \mathsf{lock}(l) \mid \mathsf{locked}(l, \mathit{Wt}, \mathit{Ot}) \mid \mathsf{ucond}(l) \mid \mathsf{cond}(l) \\
\qquad\qquad\qquad \mid \mathsf{obs}(O) \mid \mathsf{ctr}(z, n) \mid \mathsf{tic}(z) \\
\qquad\qquad\qquad \mid b \mid a_1 \wedge a_2 \mid a_1 \vee a_2 \mid a_1 * a_2 \mid a_1 \mathrel{-\!\!*} a_2 \mid \forall \alpha \mid \exists \alpha
\end{array}
$$

$$
\begin{array}{l}
\mathsf{pt} : \mathit{PredicateTables} = \mathit{Indexes} \to \mathit{Lists}(\mathit{Arguments}) \to \\
\quad \mathit{Bags}(\mathit{Obligations}) \to \mathit{Bags}(\mathit{Obligations}) \to \mathit{Assertions} \\
\mathsf{O} : \mathit{Locations} \to \mathit{Obligations} \text{ , where } \mathsf{O}((A, R, L), I, M, M') = (A, R, L) \\
\mathsf{A} : \mathit{Locations} \to \mathit{Addresses} \text{ , where } \mathsf{A}((A, R, L), I, M, M') = A \\
\mathsf{R} : \mathit{Locations} \to \mathit{Levels} \text{ , where } \mathsf{R}((A, R, L), I, M, M') = R \\
\mathsf{L} : \mathit{Locations} \to \mathit{Addresses} \text{ , where } \mathsf{L}((A, R, L), I, M, M') = L \\
\mathsf{I} : \mathit{Locations} \to \mathit{Bags}(\mathit{Obligations}) \to \mathit{Bags}(\mathit{Obligations}) \to \mathit{Assertions} \\
\quad \text{where } \mathsf{I}((A, R, L), I, M, M') = \mathsf{pt}(\mathsf{fst}(I), \mathsf{snd}(I)) \\
\mathsf{M} : \mathit{Locations} \to \mathit{Assertions}, \\
\quad \text{where } \mathsf{M}((A, R, L), I, M, M') = \mathsf{pt}(\mathsf{fst}(M), \mathsf{snd}(M), \{\!\}, \{\!\}) \\
\mathsf{M}' : \mathit{Locations} \to \mathit{Bags}(\mathit{Obligations}) \text{ , where } \mathsf{M}'((A, R, L), I, M, M') = M'
\end{array}
$$

Figure 37 Syntax of assertions

$$
\begin{array}{l}
k \in \mathit{Knowledge} ::= \mathsf{cell}(\pi, z) \mid \mathsf{ulock}(\mathit{Wt}, \mathit{Ot}) \mid \mathsf{lock} \mid \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) \mid \mathsf{ucond} \mid \mathsf{cond} \\
p \in \mathit{PermissionHeaps} = \mathit{Locations} \rightharpoonup \mathit{Knowledge} \\
\mathit{GhostIdentifications} = \mathbb{Z} \\
\mathit{gv} \in \mathit{GhostValues} ::= \mathit{Option}(\mathbb{N}) \times \mathbb{N} \\
g \in \mathit{GhostHeaps} = \mathit{GhostIdentifications} \rightharpoonup \mathit{GhostValues} \\
\mathit{Option}(A) ::= s \mid \varnothing \text{ , where } s \in A \\
\tilde{O} \in \mathit{Option}(\mathit{Bags}(\mathit{Obligations}))
\end{array}
$$

$$
\begin{array}{l}
p, \tilde{O}, g \models l \overset{\pi}{\mapsto} z \quad \Leftrightarrow \quad p(l) = \mathsf{cell}(\pi, z) \\
p, \tilde{O}, g \models \mathsf{ulock}(l,\, \mathit{Wt}, \mathit{Ot}) \quad \Leftrightarrow \quad p(l) = \mathsf{ulock}(\mathit{Wt}, \mathit{Ot}) \\
p, \tilde{O}, g \models \mathsf{lock}(l) \quad \Leftrightarrow \quad p(l) = \mathsf{lock}() \\
p, \tilde{O}, g \models \mathsf{locked}(l,\, \mathit{Wt}, \mathit{Ot}) \quad \Leftrightarrow \quad p(l) = \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) \\
p, \tilde{O}, g \models \mathsf{ucond}(l) \quad \Leftrightarrow \quad p(l) = \mathsf{ucond} \\
p, \tilde{O}, g \models \mathsf{cond}(l) \quad \Leftrightarrow \quad p(l) = \mathsf{cond} \\
p, \tilde{O}, g \models \mathsf{obs}(O) \quad \Leftrightarrow \quad \tilde{O} = O \\
p, \tilde{O}, g \models \mathsf{ctr}(z, n) \quad \Leftrightarrow \quad \exists n_1.\ g(z) = (n, n_1) \\
p, \tilde{O}, g \models \mathsf{tic}(z) \quad \Leftrightarrow \quad \exists n, \tilde{n}.\ g(z) = (\tilde{n}, n{+}1) \\
p, \tilde{O}, g \models b \quad \Leftrightarrow \quad b = \mathsf{true} \\
p, \tilde{O}, g \models a_1 \wedge a_2 \quad \Leftrightarrow \quad p, \tilde{O}, g \models a_1 \wedge p, \tilde{O}, g \models a_2 \\
p, \tilde{O}, g \models a_1 \vee a_2 \quad \Leftrightarrow \quad p, \tilde{O}, g \models a_1 \vee p, \tilde{O}, g \models a_2 \\
p, \tilde{O}, g \models a_1 * a_2 \quad \Leftrightarrow \quad \exists p_1, p_2, \tilde{O}_1, \tilde{O}_2, g_1, g_2.\ p{=}p_1 \uplus p_2 \wedge \tilde{O}{=}\tilde{O}_1 \tilde{\uplus} \tilde{O}_2 \wedge g{=}g_1 \uplus g_2 \wedge \\
\quad p_1, \tilde{O}_1, g_1 \models a_1 \wedge p_2, \tilde{O}_2, g_2 \models a_2 \\
p, \tilde{O}, g \models a_1 \mathrel{-\!\!*} a_2 \quad \Leftrightarrow \quad \forall p_1, \tilde{O}_1, g_1.\ p_1, \tilde{O}_1, g_1 \models a_1 \ \Rightarrow \\
\quad \forall p_2, \tilde{O}_2, g_2.\ p_2{=}p \uplus p_1 \wedge g_2{=}g \uplus g_1 \wedge \tilde{O}_2{=}\tilde{O} \uplus \tilde{O}_1 \ \Rightarrow \ p_2, \tilde{O}_2, g_2 \models a_2 \\
p, \tilde{O}, g \models \forall \alpha \quad \Leftrightarrow \quad \forall \hat{v}{\in}\mathit{AValues}.\ p, \tilde{O}, g \models \alpha(\hat{v}) \\
p, \tilde{O}, g \models \exists \alpha \quad \Leftrightarrow \quad \exists \hat{v}{\in}\mathit{AValues}.\ p, \tilde{O}, g \models \alpha(\hat{v})
\end{array}
$$

$$
a_1 \vdash a_2 \quad \Leftrightarrow \quad (\forall p, \tilde{O}, g.\ p, \tilde{O}, g \models a_1 \Rightarrow p, \tilde{O}, g \models a_2)
$$

Figure 38 Satisfaction relation

$$
\begin{array}{l}
O_1, O_2 \in \mathit{Bags}(A) \\
\tilde{O}_1, \tilde{O}_2 \in \mathit{Option}(A) \\
gv_1, gv_2 \in \mathit{GhostValues} \\
g_1, g_2 \in \mathit{GhostIdentifications} \rightharpoonup \mathit{GhostValues} \\
k_1, k_2 \in \mathit{Knowledge} \\
p_1, p_2 \in \mathit{PermissionHeaps}
\end{array}
$$

$$O_1 \uplus O_2 = \lambda v.\ O_1(v) + O_2(v)$$

$$\tilde{O}_1 \uplus \tilde{O}_2 = \begin{cases} \tilde{O}_1 & \text{if } \tilde{O}_2{=}\varnothing \\ \tilde{O}_2 & \text{if } \tilde{O}_1{=}\varnothing \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\tilde{O}_1 \tilde{\uplus} \tilde{O}_2 = \begin{cases} \tilde{O}_1 & \text{if } \tilde{O}_2 = \varnothing \\ \tilde{O}_2 & \text{if } \tilde{O}_1 = \varnothing \\ \tilde{O}_1 \uplus \tilde{O}_2 & \text{otherwise} \end{cases}$$

$$gv_1 \tilde{\uplus} gv_2 = \begin{cases} (\tilde{m}_1 \uplus \tilde{m}_2, n_1{+}n_2) & \text{if } gv_1 = (\tilde{m}_1, n_1) \wedge gv_2 = (\tilde{m}_2, n_2) \wedge \\ & \quad (\tilde{m}_1 \uplus \tilde{m}_2 = n \Rightarrow n \geqslant n_1{+}n_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$g_1 \uplus g_2 = \begin{cases} \lambda l.\ g_1(l) \tilde{\uplus} g_2(l) & \text{if } \exists g.\ \forall l.\ g(l) = g_1(l) \tilde{\uplus} g_2(l) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$k_1 \uplus k_2 = \begin{cases} \mathsf{cell}(\pi{+}\pi', z) & \text{if } k_1{=}\mathsf{cell}(\pi, z) \wedge k_2{=}\mathsf{cell}(\pi', z) \wedge \pi{+}\pi' \leqslant 1 \\ \mathsf{lock} & \text{if } k_1{=}\mathsf{lock} \wedge k_2{=}\mathsf{lock} \\ \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) & \text{if } k_1{=}\mathsf{lock} \wedge k_2 = \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) \\ \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) & \text{if } k_1 = \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) \wedge k_2{=}\mathsf{lock} \\ \mathsf{cond} & \text{if } k_1{=}\mathsf{cond} \wedge k_2{=}\mathsf{cond} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$p_1 \uplus p_2 = \begin{cases} \lambda l.\ p_1(l) \tilde{\uplus} p_2(l) & \text{if } \exists p.\ \forall l.\ p(l) = p_1(l) \tilde{\uplus} p_2(l) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Figure 39 Operations on ghost resources

$$
\begin{aligned}
&\mathsf{pheap\_heap}(p, h) \Leftrightarrow \\
&(\forall z.\ (\forall l.\ \mathsf{A}(l) = z \Rightarrow p(l) = \varnothing) \Rightarrow h(z) = \varnothing) \text{ and} \\
&\forall l. \\
&\quad p(l) = \varnothing \text{ or} \\
&\quad \forall z.\ (\exists \pi.\ p(l) = \mathsf{cell}(\pi, z)) \Rightarrow h(\mathsf{A}(l)) = z \text{ and} \\
&\quad (\exists O_1, O_2.\ p(l) = \mathsf{ulock}(O_1, O_2)) \Rightarrow h(\mathsf{A}(l)) = 1 \text{ and} \\
&\quad p(l) = \mathsf{lock} \Rightarrow h(\mathsf{A}(l)) = 1 \text{ and} \\
&\quad (\exists O_1, O_2.\ p(l) = \mathsf{locked}(O_1, O_2)) \Rightarrow h(\mathsf{A}(l)) = 0 \text{ and} \\
&\quad p(l) = \mathsf{cond} \Rightarrow h(\mathsf{A}(l)) \neq \varnothing \text{ and} \\
&\quad p(l) = \mathsf{ucond} \Rightarrow h(\mathsf{A}(l)) \neq \varnothing
\end{aligned}
$$

Figure 40 Permission heaps corresponding to concrete heaps

D.2 Syntax and Semantics of Assertions

The syntax of assertions is defined in Figure 37[14]. Note that a *location* l of an object o consists of the *obligation* of o (if o is a lock or a CV); the lock invariant of o (if o is a lock), denoted by $\mathsf{I}(l)$; and the permissions and the obligations which are transferred through a notification on o (if o is a CV), denoted by $\mathsf{M}(l)$ and $\mathsf{M}'(l)$ respectively. Also note that permissions described by invariants of locks as well as permissions which are transferred through notifications are specified through an index (as well as the required arguments) pointing to a table in which each element is a function that given a list of arguments returns an assertion. This makes it possible to quantify over locations in assertions[15]. The obligation of a location l, denoted by $\mathsf{O}(l)$, consists of the address of that location, denoted by $\mathsf{A}(l)$, as well as other ghost information such as the level of l, denoted by $\mathsf{R}(l)$; and the lock associated with l (if l is the location of a CV), denoted by $\mathsf{L}(l)$.

These assertions describe some ghost resources, namely p, O, and g that keep track of heap locations, obligations, and ghost counters, respectively, shown in Figure 38, where some operations and relations defined on these resources are shown in Figures 39, 40.

D.3 Weakest Precondition of Commands

The weakest precondition of a command c for $n{>}0$ steps w.r.t. a postcondition Q (with a given predicate table, specified by pt), denoted by $\mathsf{wp}_{n,pt}(c, Q)$ is defined in Figures 41 and 42. Note that $\mathsf{wp}(c, Q)_{0,pt} = \mathsf{true}$. Also note that for the sake of simplicity the index pt is elided. Having this definition, we define the weakest precondition of a context and the

[11] The soundness proof of the second mechanism, machine-checked in Coq, can be found in [17].

[12] Note that one way to formalize the precise approach of Section 3 would be to define assertions as functions from ghost information to separating conjunctions of chunks. In the soundness proof, one would track these as partial functions whose domain is the set of allocated addresses. The functions passed into the assertions would be *totalizations* of these partial functions. An assertion is true if it is true for all totalizations of the functions.

[13] Note that $\mathsf{new\_int}(n)$ allocates n consecutive memory locations and returns the address of the first one.

[14] Note that we use a shallow embedding: assertions have no variables; to model quantifications, we use meta-level functions from values to assertions.

[15] An alternative approach is to use a step-indexed domain of assertions, as in Iris [27]. There, ▶ *Assertions* could be used instead of *Indexes* × *Lists*(*Arguments*), where ▶ is Iris's *guard* for *guarded recursive definitions*.

$$\mathsf{wp} \in \mathit{WeakestPreconditions} = \mathit{Commands} \to (\mathbb{Z} \to \mathit{Assertions}) \to \mathbb{N} \to \mathit{PredicateTables} \to \mathit{Assertions}$$

$$\mathsf{wp}_n(\mathsf{val}(e), Q) = Q([\![e]\!])$$
$$\mathsf{wp}_n(\mathsf{new\_lock}, Q) = \forall z.\ \exists r.\ \mathsf{ulock}(((z,r,z),(0,[]),(0,[]),\{\!\}),\{\!\},\{\!\}) \mathrel{-\!\!*} Q(z)$$
$$\mathsf{wp}_n(\mathsf{acquire}(e), Q) = \exists O, l.\ (\mathsf{lock}(l) * \mathsf{obs}(O) \wedge \mathsf{O}(l) \prec O \wedge \mathsf{A}(l) = [\![e]\!]) * (\forall\, Wt, Ot.\ (\mathsf{obs}(O \uplus \{\!\{\mathsf{O}(l)\}\!\}) * \mathsf{locked}(l,\, Wt, Ot) * \mathsf{I}(l)(\,Wt, Ot)) \mathrel{-\!\!*} Q(\mathsf{tt}))$$
$$\mathsf{wp}_n(\mathsf{waiting4lock}(e), Q) = \mathsf{wp}_n(\mathsf{acquire}(e), Q)$$
$$\mathsf{wp}_n(\mathsf{release}(e), Q) = \exists\, Wt, Ot, O, l.\ (\mathsf{locked}(l,\, Wt, Ot) * \mathsf{I}(l)(\,Wt, Ot) * \mathsf{obs}(O \uplus \{\!\{\mathsf{O}(l)\}\!\}) \wedge \mathsf{A}(l) = [\![e]\!]) * ((\mathsf{lock}(l) * \mathsf{obs}(O)) \mathrel{-\!\!*} Q(\mathsf{tt}))$$
$$\mathsf{wp}_n(\mathsf{new\_cvar}, Q) = \forall z.\ \exists r.\ \mathsf{ucond}(((z,r,z),(0,[]),(0,[]),\{\!\})) \mathrel{-\!\!*} Q(z)$$
$$\mathsf{wp}_n(\mathsf{wait}(e_1, e_2), Q) = \exists\, Wt, Ot, O, v, l.\ (\mathsf{cond}(v) * \mathsf{locked}(l,\, Wt, Ot) * \mathsf{I}(l)(\,Wt \uplus \{\!\{\mathsf{A}(v)\}\!\}, Ot) * \mathsf{obs}(O \uplus \{\!\{\mathsf{O}(l)\}\!\}) \wedge \mathsf{L}(v){=}\mathsf{A}(l) \wedge \mathsf{O}(v) \prec O \wedge \mathsf{O}(l) \prec O \uplus \mathsf{M}'(v) \wedge \mathsf{enoughObs}(v,\, Wt \uplus \{\!\{\mathsf{A}(v)\}\!\}, Ot) \wedge \mathsf{A}(v) = [\![e_1]\!] \wedge \mathsf{A}(l) = [\![e_2]\!]) * (\forall\, Wt', Ot'.\ (\mathsf{cond}(v) * \mathsf{locked}(l,\, Wt', Ot') * \mathsf{I}(l)(\,Wt', Ot') * \mathsf{obs}(O \uplus \{\!\{\mathsf{O}(l)\}\!\} \uplus \mathsf{M}'(v)) * \mathsf{M}(v)) \mathrel{-\!\!*} Q(\mathsf{tt}))$$
$$\mathsf{wp}_n(\mathsf{waiting4cvar}(e_1, e_2), Q) = \exists O, v, l.\ (\mathsf{cond}(v) * \mathsf{lock}(l) * \mathsf{obs}(O) \wedge \mathsf{O}(v) \prec O \wedge \mathsf{O}(l) \prec O \uplus M' \wedge \mathsf{L}(v){=}\mathsf{A}(l) \wedge \mathsf{A}(v){=}[\![e_1]\!] \wedge \mathsf{A}(l) = [\![e_2]\!]) * (\forall\, Wt, Ot.\ (\mathsf{cond}(v) * \mathsf{locked}(l,\, Wt, Ot) * \mathsf{I}(l)(\,Wt, Ot) * \mathsf{obs}(O \uplus \{\!\{\mathsf{O}(l)\}\!\} \uplus \mathsf{M}'(v)) * \mathsf{M}(v)) \mathrel{-\!\!*} Q(\mathsf{tt}))$$
$$\mathsf{wp}_n(\mathsf{notify}(e), Q) = \exists\, Wt, Ot, O, v.\ (\mathsf{cond}(v) * \mathsf{locked}(\mathsf{L}(v),\, Wt, Ot) * \mathsf{obs}(O \uplus (0{<}\,Wt(\mathsf{A}(v))\ ?\ \mathsf{M}'(v) : \{\!\})) * (\,Wt(\mathsf{A}(v)){=}0 \vee \mathsf{M}(v)) \wedge \mathsf{A}(v) = [\![e]\!]) * ((\mathsf{cond}(v) * \mathsf{locked}(\mathsf{L}(v),\, Wt{-}\{\!\{\mathsf{A}(v)\}\!\}, Ot) * \mathsf{obs}(O)) \mathrel{-\!\!*} Q(\mathsf{tt}))$$
$$\mathsf{wp}_n(\mathsf{notifyAll}(e), Q) = \exists\, Wt, Ot, O, v, l.\ (\mathsf{cond}(v) * \mathsf{locked}(l,\, Wt, Ot) * (\mathop{*}_{i:=1}^{Wt(v)} \mathsf{M}(v)) \wedge \mathsf{M}'(v) = \{\!\} \wedge \mathsf{A}(v){=}[\![e]\!] \wedge \mathsf{L}(v){=}\mathsf{A}(l)) * ((\mathsf{cond}(v) * \mathsf{locked}(l,\, Wt[\mathsf{A}(v){:=}0], Ot)) \mathrel{-\!\!*} Q(\mathsf{tt}))$$
$$\mathsf{wp}_n(\mathsf{let}(x, c_1, c_2), Q) = \mathsf{wp}_{n-1}(c_1, \lambda z.\ \mathsf{wp}_{n-1}(c_2[z/x], Q))$$
$$\mathsf{wp}_n(\mathsf{fork}(c), Q) = \exists O_1, O_2.\ \mathsf{obs}(O_1 \uplus O_2) * (\mathsf{obs}(O_1) \mathrel{-\!\!*} Q(\mathsf{tt})) * (\mathsf{obs}(O_2) \mathrel{-\!\!*} \mathsf{wp}_{n-1}(c, \lambda\_.\ \mathsf{obs}(\{\!\})))$$
$$\mathsf{wp}_n(\mathsf{if}(c, c_1, c_2), Q) = \mathsf{wp}_{n-1}(c, (\lambda z.\ 0{<}z\ ?\ \mathsf{wp}_{n-1}(c_1, Q) : \mathsf{wp}_{n-1}(c_2, Q)))$$
$$\mathsf{wp}_n(\mathsf{while}(c, c_1), Q) = \mathsf{wp}_{n-1}(c, (\lambda z.\ z{\leqslant}0\ ?\ Q(\mathsf{tt}) : \mathsf{wp}_{n-1}(c_1, (\lambda\_.\ \mathsf{wp}_{n-1}(\mathsf{while}(c, c_1), Q)))))$$

Figure 41 Weakest precondition, where tt stands for 0 (part one of two).

$\mathsf{wp}_{n,pt}(\mathsf{nop}) =$

as g_initl	$(\exists\, Wt, Ot, O, a, r, I.\ \mathsf{ulock}(((a,r,a),(0,[]),(0,[]),\{\!\}),\, Wt, Ot) * pt(I)(\,Wt, Ot) * \mathsf{obs}(O) *$ $((\mathsf{lock}(((a,r,a), I, (0,[]),\{\!\})) * \mathsf{obs}(O)) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee$
as g_initc	$(\exists\, Wt, Ot, a, r, l, M, M'.\ \mathsf{ucond}((a,r,a),(0,[]),(0,[]),\{\!\}) * \mathsf{ulock}(l,\, Wt, Ot) *$ $((\mathsf{cond}((a,r,\mathsf{A}(l)),(0,[]), M, M') * \mathsf{ulock}(l,\, Wt, Ot)) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee$
as g_load	$(\exists v, l.\ (\mathsf{cond}(v) * \mathsf{ulock/locked}(l,\, Wt, Ot) * \mathsf{obs}(O) \wedge \mathsf{L}(v){=}\mathsf{A}(l)) *$ $((\mathsf{cond}(v) * \mathsf{ulock/locked}(l,\, Wt, Ot{\uplus}\{\!\{\mathsf{A}(v)\}\!\}) * \mathsf{obs}(O{\uplus}\{\!\{\mathsf{O}(v)\}\!\})) \mathrel{-\!\!*} Q(\mathsf{tt})))$
as g_discharge	$(\exists v, l.\ (\mathsf{cond}(v) * \mathsf{ulock/locked}(l,\, Wt, Ot) * \mathsf{obs}(O) \wedge \mathsf{enoughObs}(v,\, Wt, Ot{-}\{\!\{\mathsf{A}(v)\}\!\}) \wedge$ $\mathsf{L}(v){=}\mathsf{A}(l)) * ((\mathsf{cond}(v) * \mathsf{ulock/locked}(l,\, Wt, Ot{-}\{\!\{\mathsf{A}(v)\}\!\}) * \mathsf{obs}(O{-}\{\!\{\mathsf{O}(v)\}\!\})) \mathrel{-\!\!*} Q(\mathsf{tt})))\vee$
as g_new_ctr	$(\forall gv.\ \mathsf{ctr}(gv, 0) \mathrel{-\!\!*} Q(\mathsf{tt})) \vee$
as g_inc	$(\exists n, gv.\ \mathsf{ctr}(gv, n) * ((\mathsf{ctr}(gv, n{+}1) * \mathsf{tic}(gv)) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee$
as g_dec	$(\exists n, gv.\ \mathsf{ctr}(gv, n) * \mathsf{tic}(gv) * (\mathsf{ctr}(gv, n{-}1) \mathrel{-\!\!*} Q(\mathsf{tt})))$

Figure 42 Weakest precondition (part two of two).

weakest precondition of a command-context as shown in Definitions 17 and 18. Having these definitions, we can prove some auxiliary lemmas, shown in Lemmas 20, 21, 22, 23, and 24, which are used to prove Theorem 31.

▶ **Definition 17** (Weakest Precondition of a Context)**.**

$$\mathsf{wpx}_n(\xi) = \begin{cases} \lambda\_.\ \mathsf{obs}(\{\!\}) & \text{if } \xi{=}\mathsf{done} \\ \lambda z.\ \mathsf{wp}_n(c[z/x], \mathsf{wpx}_n(\xi')) & \text{if } \xi{=}\mathsf{let}'(x, c, \xi') \\ \lambda z.\ 0{<}z\ ?\ \mathsf{wp}_n(c_1, (\mathsf{wpx}_n(\xi'))) : \mathsf{wp}_n(c_2, (\mathsf{wpx}_n(\xi'))) & \text{if } \xi{=}\mathsf{if}'(c_1, c_2, \xi') \end{cases}$$

▶ **Definition 18** (Weakest Precondition of a command-context)**.**

$$\mathsf{wpcx}_n(c, \xi) = \mathsf{wp}_n(c, \mathsf{wpx}_n(\xi))$$

▶ **Lemma 19** (Weakening Postcondition)**.**

$$p, \tilde{O}, g \models \mathsf{wp}_n(c, Q) \wedge (\forall z.\ Q(z) \vdash Q'(z)) \Rightarrow \forall n'{\leqslant}n.\ p, \tilde{O}, g \models \mathsf{wp}_{n'}(c, Q')$$

Proof. By induction on n and case analysis of c. ◀

▶ **Lemma 20** (Weakest Precondition of Wait)**.**

$$\begin{array}{l} \forall n, e_1, e_2, \xi, p, O, g.\quad p, O, g \models \mathsf{wpcx}_n(\mathsf{wait}(e_1, e_2), \xi) \Rightarrow \\ \exists p_1, p_2, g_1, g_2, O_1, v, l,\, Wt, Ot.\ p{=}p_1{\uplus}p_2 \wedge O{=}O_1{\uplus}\{\!\{\mathsf{O}(l)\}\!\} \wedge g{=}g_1{\uplus}g_2 \wedge \\ \mathsf{A}(v) = [\![e_1]\!] \wedge \mathsf{A}(l) = [\![e_2]\!] \wedge p_1(l){=}\mathsf{locked}(\,Wt, Ot) \wedge p_1(v){=}\mathsf{cond} \wedge \\ p_2, \varnothing, g_2 \models \mathsf{I}(l)(\,Wt{\uplus}\{\!\{\mathsf{A}(v)\}\!\}, Ot) \wedge \mathsf{O}(v) \prec O_1 \wedge \mathsf{O}(l) \prec O_1{\uplus}\mathsf{M}'(v) \wedge \mathsf{L}(v){=}\mathsf{A}(l) \wedge \\ \mathsf{enoughObs}(v,\, Wt{\uplus}\{\!\{\mathsf{A}(v)\}\!\}, Ot) \wedge \\ p_1[l{:=}\mathsf{lock}], O_1, g_1 \models \mathsf{wpcx}_n(\mathsf{waiting4cvar}(e_1, e_2), \xi) \end{array}$$

▶ **Lemma 21** (Weakest Precondition of Notify).

$$\begin{array}{l}\forall n, e, \xi, p, O, g.\quad p, O, g \models \mathsf{wpcx}_n(\mathsf{notify}(e), \xi) \Rightarrow \exists p_1, p_M, g_1, g_M, O_1, v, l,\, Wt, Ot.\\ p{=}p_1 \uplus p_M \wedge g{=}g_1 \uplus g_M \wedge O{=}O_1 \uplus (0{<}\,Wt(\mathsf{A}(v))\ ?\ \mathsf{M}'(v) : \{\!\}) \\ \wedge\ \mathsf{A}(v) = [\![e]\!] \wedge \mathsf{L}(v) = \mathsf{A}(l) \wedge p_1(v){=}\mathsf{cond} \wedge p_1(l){=}\mathsf{locked}(\,Wt, Ot) \wedge \\ (0{<}\,Wt(\mathsf{A}(v))\ ?\ p_M, \varnothing, g_M \models \mathsf{M}(v) : (p_M{=}\mathbf{0} \wedge g_M{=}\mathbf{0})) \wedge \\ p_1[l{:=}\mathsf{locked}(\,Wt{-}\{\!|\mathsf{A}(v)|\!\}, Ot)], O_1, g_1 \models \mathsf{wpcx}_n(\mathsf{tt}, \xi)\end{array}$$

▶ **Lemma 22** (Weakest Precondition of waiting4cvar).

$$\begin{array}{l}\forall n, e_1, e_2, \xi, p, O, g.\quad p, O, g \models \mathsf{wpcx}_n(\mathsf{waiting4cvar}(e_1, e_2), \xi) \Rightarrow \\ \exists v, l.\ p(v){=}\mathsf{cond} \wedge (p(l){=}\mathsf{lock} \vee \exists\, Wt, Ot.\ p(l){=}\mathsf{locked}(\,Wt, Ot)) \wedge \mathsf{L}(v){=}\mathsf{A}(l) \wedge \mathsf{O}(v) \prec O \\ \wedge\ \mathsf{O}(l) \prec O \uplus \mathsf{M}'(v) \wedge \mathsf{A}(v) = [\![e_1]\!] \wedge \mathsf{A}(l) = [\![e_2]\!] \wedge \\ \forall p_M, g_M.\ p_M, \varnothing, g_M \models \mathsf{M}(v) \Rightarrow p \uplus p_M, O \uplus \mathsf{M}'(v), g \uplus g_M \models \mathsf{wpcx}_n(\mathsf{waiting4lock}(e_2), \xi)\end{array}$$

▶ **Lemma 23** (Weakest Precondition of g_discharge).

$$\begin{array}{l}\forall n, \xi, p, O, g.\quad p, O, g \models \mathsf{wpcx}_n(\mathsf{g\_discharge}, \xi) \Rightarrow \\ \exists O_1,\, Wt, Ot, v, l.\ O{=}O_1 \uplus \{\!|\mathsf{O}(v)|\!\} \wedge p(l){=}\mathsf{locked}(\,Wt, Ot) \wedge p(v){=}\mathsf{cond} \wedge \\ \mathsf{enoughObs}(v,\, Wt, Ot{-}\{\!|\mathsf{A}(v)|\!\}) \wedge \mathsf{L}(v){=}\mathsf{A}(l) \wedge \\ p[l{:=}\mathsf{locked}(\,Wt, Ot{-}\{\!|\mathsf{A}(v)|\!\})], O_1, g \models \mathsf{wpcx}_n(\mathsf{tt}, \xi)\end{array}$$

▶ **Lemma 24** (Weakest Precondition of fork).

$$\begin{array}{l}\forall n, c, \xi, p, O, g.\quad p, O, g \models \mathsf{wpcx}_n(\mathsf{fork}(c), \xi) \Rightarrow \\ \exists p_1, p_2, g_1, g_2, O_1, O_2.\ p{=}p_1 \uplus p_2 \wedge g{=}g_1 \uplus g_2 \wedge O{=}O_1 \uplus O_2 \wedge \\ p_1, O_1, g_1 \models \mathsf{wpcx}_n(\mathsf{tt}, \xi) \wedge p_2, O_2, g_2 \models \mathsf{wp}_{n-1}(c, \lambda\_.\mathsf{obs}(\{\!\})) \end{array}$$

▶ **Lemma 25** (Frame in Weakest Precondition).

$$\forall n, c, Q, F, p, \tilde{O}, g.\quad p, \tilde{O}, g \models \mathsf{wp}_n(c, Q) * F \Rightarrow \forall n' {\leqslant} n.\ p, \tilde{O}, g \models \mathsf{wp}_{n'}(c, (\lambda z.\ Q(z) * F))$$

Proof. By induction on n and case analysis of c. ◀

D.4 Correctness of Commands

We define *correctness of commands*, as shown in Definition 26, ensuring that each proposed proof rule, where $\mathsf{correct}_{pt}(P, c, Q)$ is abbreviated as $\{P\}\ c\ \{Q\}$, respects the definition of the weakest precondition. Having this definition we prove the proposed proof rules, ensuring deadlock freedom of importer channels, as well as some other necessary proof rules shown in Theorems 27, 28, and 29.

▶ **Definition 26** (Correctness of Commands). *A command is correct w.r.t a precondition P and a postcondition Q if and only if P implies the weakest precondition of that command w.r.t Q.*

$$\mathsf{correct}_{pt}(P, c, Q) \Leftrightarrow \forall n.\ P \Rightarrow \mathsf{wp}_{n,pt}(c, Q)$$

▶ **Theorem 27** (Rule Sequential Composition).

$$\mathsf{correct}(P, c_1, Q) \wedge (\forall z.\ \mathsf{correct}(Q(z), c_2[z/x], R)) \Rightarrow \mathsf{correct}(P, \mathsf{let}(x, c_1, c_2), R)$$

▶ **Theorem 28** (Rule Consequence).

$$\mathsf{correct}(P, c, Q) \wedge (P' \vdash P) \wedge (\forall z.\ Q(z) \vdash Q'(z)) \Rightarrow \mathsf{correct}(P', c, Q')$$

▶ **Theorem 29** (Rule Frame).

$$\mathsf{correct}(P, c, Q) \Rightarrow \mathsf{correct}(P * F, c, \lambda z.\ Q(z) * F)$$

As previously mentioned, since in this formalization ghost information is associated with lock and condition variable addresses via the lock and cond permission rather than via global function, we provide a new version of the proof rules, proposed in Section 3, regarding this formalization as shown in Figure 43.

D.5 Validity of a Configuration

We define *validity of a configuration*, shown in Definition 30, and prove that 1) starting from a valid configuration, all the subsequent configurations of the execution are also valid (Theorem 31), 2) a valid configuration is not deadlocked (Theorem 32), and 3) if a program c is verified by the proposed proof rules, where the verification starts from an empty bag of obligations and ends with such a bag too, then the initial configuration, where the heap is empty, denoted by $\mathbf{0}{=}\lambda\_.\varnothing$, and there is only one thread with the command c (and a context done), is a valid configuration (Theorem 34).

▶ **Definition 30** (Validity of a Configuration). *A configuration (t, h) is valid for n steps, denoted by $\mathsf{valid}_n(t, h)$, if there exist a list of* augmented threads *A, consisting of the identification (id), the program (c), the context (ξ), the permission heap (p), the ghost resource heap (g) and the obligations (O) associated with each thread; a list of lock-invariant pairs $Linv$, storing the locks which are not held along with their invariants; three permission heaps p_i (associated with the invariants of the locks which are not held), p_l (the part of the permission heap which is leaked), and p_A (the union of all permission heaps in A and p_i as well as p_l); three ghost resource heaps g_i (associated with the invariants of the locks which are not held), g_l (the part of the ghost resource heap which is leaked), g_A (the union of all ghost resource heaps in A and g_i as well as g_l); and locs (the set of locations for which a memory has been allocated), such that all of the following conditions hold:*

1. $\forall id, c, \xi.\ t(id) = (c; \xi) \Leftrightarrow \exists p, O, g.\ (id, c, \xi, p, O, g) \in A$
2. $\forall (id_1, c_1, \xi_1, p_1, O_1, g_1) \in A, (id_2, c_2, \xi_2, p_2, O_2, g_2) \in A.\ id_1 = id_2 \Rightarrow$
   $(id_1, c_1, \xi_1, p_1, O_1, g_1) = (id_2, c_2, \xi_2, p_2, O_2, g_2)$
3. $p_A = p_i \uplus p_l \uplus \underset{(id,c,\xi,p,O,g)\in A}{\uplus} p \quad \wedge \quad g_A = g_i \uplus g_l \uplus \underset{(id,c,\xi,p,O,g)\in A}{\uplus} g$
4. $\forall l_1, l_2.\ p_A(l_1) \neq \varnothing \wedge p_A(l_2) \neq \varnothing \wedge \mathsf{A}(l_1) = \mathsf{A}(l_1) \Rightarrow l_1 = l_2$
5. $\forall l.\ p_A(l) \neq \varnothing \Leftrightarrow l \in locs$
6. $\mathsf{pheap\_heap}(p_A, h)$
7. $p_i, \varnothing, g_i \models \underset{(l,inv)\in Linv}{*} inv$
8. $p_A(l) = \mathsf{lock} \wedge \neg \mathsf{held}_h(l) \Rightarrow (l, \mathsf{I}(l)(Wt_{l,A}, Ot_{l,A})) \in Linv$
9. $(l, inv) \in Linv \Rightarrow p_A(l) = \mathsf{lock} \wedge \neg \mathsf{held}_h(l)$
10. $\forall o \in O_A.\ \exists l.\ \mathsf{O}(l){=}o \wedge (p_A(l){=}\mathsf{cond} \vee p_A(l){=}\mathsf{lock} \vee \exists Wt, Ot.\ p_A(l){=}\mathsf{locked}(Wt, Ot))$
11. $p_A(l){=}\mathsf{ulock}(Wt, Ot) \vee p_A(l){=}\mathsf{locked}(Wt, Ot) \Rightarrow Wt{=}Wt_{l,A} \wedge Ot{=}Ot_{l,A}$
12. $p_A(l){=}\mathsf{lock} \vee p_A(l){=}\mathsf{ulock}(Wt, Ot) \vee p_A(l){=}\mathsf{locked}(Wt, Ot) \Rightarrow \mathsf{held}_h(l) \Rightarrow l \in O_A$
13. $\forall (id, c, \xi, p, O, g){\in}A.$
    a. $p, O, g \models \mathsf{wpcx}_n(c, \xi)$
    b. $c = \mathsf{waiting4cvar}(e_1, e_2) \Rightarrow \mathsf{enoughObs}([\![e_1]\!], Wt_{[\![e_2]\!],A}, Ot_{[\![e_2]\!],A})$

where

- $O_A = \underset{(id,c,\xi,p,O,g)\in A}{\uplus} O$

NewLock
$\{\mathsf{true}\}\ \mathsf{newlock}\ \{\lambda a.\ \mathsf{ulock}(((a, r, a), (0, []), (0, []), \{\!\}), \{\!\}, \{\!\})\}$

InitLock
$\{\mathsf{ulock}(((a, r, a), (0, []), (0, []), \{\!\}), Wt, Ot) * pt(I_{index}, I_{args})(Wt, Ot) * \mathsf{obs}(O)\}\ \mathsf{nop}$
$\{\lambda\_.\ \mathsf{lock}(((a, r, a), (I_{index}, I_{args}), (0, []), \{\!\})) * \mathsf{obs}(O)\}$

Acquire
$\{\mathsf{lock}(l) * \mathsf{obs}(O) \wedge \mathsf{O}(l) \prec O \wedge \mathsf{A}(l) = a_l\}\ \mathsf{acquire}(a_l)$
$\{\lambda\_.\ \exists Wt, Ot.\ \mathsf{locked}(l, Wt, Ot) * \mathsf{I}(l)(Wt, Ot) * \mathsf{obs}(O \uplus \{\!|\mathsf{O}(l)|\!\})\}$

Release
$\{\mathsf{locked}(l, Wt, Ot) * \mathsf{I}(l)(Wt, Ot) * \mathsf{obs}(O \uplus \{\!|\mathsf{O}(l)|\!\}) \wedge \mathsf{A}(l) = a_l\}\ \mathsf{release}(a_l)$
$\{\lambda\_.\ \mathsf{lock}(l) * \mathsf{obs}(O)\}$

NewCV
$\{\mathsf{true}\}\ \mathsf{new\_cvar}\ \{\lambda a.\ \mathsf{ucond}(((a, r, a), (0, []), (0, []), \{\!\}))\}$

InitCV
$\{\mathsf{ucond}((a, r, a), (0, []), (0, []), \{\!\}) * \mathsf{ulock}(l, Wt, Ot)\}\ \mathsf{nop}$
$\{\lambda\_.\ \mathsf{cond}((a, r, \mathsf{A}(l)), (0, []), (M_{index}, M_{args}), M') * \mathsf{ulock}(l, Wt, Ot)\}$

Wait
$\{\mathsf{cond}(v) * \mathsf{locked}(l, Wt, Ot) * \mathsf{I}(l)(Wt \uplus \{\!|\mathsf{A}(v)|\!\}, Ot) * \mathsf{obs}(O \uplus \{\!|\mathsf{O}(l)|\!\}) \wedge \mathsf{A}(v) = a_v \wedge \mathsf{A}(l) = a_l$
$\wedge\ \mathsf{A}(l){=}\mathsf{L}(v) \wedge \mathsf{O}(v) \prec O \wedge \mathsf{O}(l) \prec O \uplus \mathsf{M}'(v) \wedge \mathsf{enoughObs}(\mathsf{A}(v), Wt \uplus \{\!|\mathsf{A}(v)|\!\}, Ot)\}\ \mathsf{wait}(a_v, a_l)$
$\{\lambda\_.\ \mathsf{cond}(v) * \mathsf{obs}(O \uplus \{\!|\mathsf{O}(l)|\!\} \uplus \mathsf{M}'(v)) * \exists Wt', Ot'.\ \mathsf{locked}(l, Wt', Ot') * \mathsf{I}(l)(Wt', Ot') * \mathsf{M}(v)\}$

Notify
$\{\mathsf{obs}(O \uplus (0 < Wt(\mathsf{A}(v))\ ?\ \mathsf{M}'(v) : \{\!\})) * \mathsf{cond}(v) * \mathsf{locked}(l, Wt, Ot) * (Wt(\mathsf{A}(v)){=}0 \vee \mathsf{M}(v)) \wedge$
$\mathsf{A}(l){=}\mathsf{L}(v) \wedge \mathsf{A}(v) = a_v\}\ \mathsf{notify}(a_v)\ \{\lambda\_.\ \mathsf{obs}(O) * \mathsf{cond}(v) * \mathsf{locked}(l, Wt{-}\{\!|\mathsf{A}(v)|\!\}, Ot)\}$

NotifyAll
$\{\mathsf{cond}(v) * \mathsf{locked}(l, Wt, Ot) * (\mathop{*}_{i:=1}^{Wt(\mathsf{A}(v))} \mathsf{M}(v)) \wedge \mathsf{M}'(v){=}\{\!\} \wedge \mathsf{A}(l){=}\mathsf{L}(v) \wedge \mathsf{A}(v) = a_v\}$
$\mathsf{notifyAll}(a_v)\ \{\lambda\_.\ \mathsf{cond}(v) * \mathsf{locked}(l, Wt[v{:=}0], Ot)\}$

ChargeObligation
$\{\mathsf{obs}(O) * \mathsf{cond}(v) * \mathsf{ulock/locked}(l, Wt, Ot) \wedge \mathsf{A}(l){=}\mathsf{L}(v)\}\ \mathsf{nop}$
$\{\lambda\_.\ \mathsf{obs}(O \uplus \{\!|\mathsf{O}(v)|\!\}) * \mathsf{cond}(v) * \mathsf{ulock/locked}(l, Wt, Ot \uplus \{\!|\mathsf{A}(v)|\!\})\}$

DischargeObligation
$\{\mathsf{obs}(O) * \mathsf{cond}(v) * \mathsf{ulock/locked}(l, Wt, Ot) \wedge \mathsf{enoughObs}(\mathsf{A}(v), Wt, Ot{-}\{\!|\mathsf{A}(v)|\!\})$
$\wedge\ \mathsf{A}(l){=}\mathsf{L}(v)\}\ \mathsf{nop}\ \{\lambda\_.\ \mathsf{obs}(O{-}\{\!|\mathsf{O}(v)|\!\}) * \mathsf{ulock/locked}(l, Wt, Ot{-}\{\!|\mathsf{A}(v)|\!\})\}$

Figure 43 Proof rules verifying deadlock-freedom of importer monitors, where ghost information is associated with lock and channel addresses via lock and cond permissions.

- $Ot_{l,A} = \lambda v.\ \mathsf{L}(v){=}l\ ?\ O_A(v) : 0$, *and* $Wt_{l,A} = \biguplus_{(id,c,\xi,p,O,g)\in A \wedge \mathsf{waiting\_for}_h(c)=v \wedge \mathsf{L}(v)=l} \{\!|v|\!\}$
- $\mathsf{waiting\_for}_h(c)$ *returns the object for which* c *is waiting, if any, i.e.*

$$\mathsf{waiting\_for}_h(c,h) = \begin{cases} [\![e_1]\!] & \text{if } c{=}\mathsf{waiting4cond}(e_1,e_2) \\ [\![e]\!] & \text{if } c{=}\mathsf{waiting4lock}(e) \wedge h([\![e]\!]){\neq}1 \\ \varnothing & \text{otherwise} \end{cases}$$

- $\mathsf{held}_h(l)$ *returns true if and only if the lock* l *is held, i.e.* $\mathsf{held}_h(l) \Rightarrow h(\mathsf{A}(l)) \neq 1$

Each item in this definition ensures some properties as follows: (1) and (2) ensure that the list of augmented threads A is correctly associated with the thread tables t, (3) ensures that the union of all permission heaps as well as the union of all ghost resource heaps in A are defined, (4) ensures that any two allocated locations which have the same address are equal, (5) *locs* is the set of locations for which a memory has been allocated (having this set it is possible to map addresses to their ghost information) (6) ensures that p_A corresponds to the concrete heap h, (7) ensures that p_i and g_i model the separating conjunction of the invariants of the locks which are not held, and these invariants do not assert any obligation, (8) ensures that any lock which is not held along with its invariant exist in Linv, (9) ensures that the locks in Linv are not held, (10) ensures that any obligation in A is associated with the address of a condition variable or a lock which has been initialized, (11) ensures that the parameters *Wt* and *Ot* stored in the permissions ulock and locked of any lock store the total number of waiting threads and obligations of the condition variables associated with that lock, respectively, (12) ensures that any lock l which is held exists in the union of the obligations of A, (13-a) the permission heap, the obligations, and the ghost resource heap of each thread model the weakest precondition of the command of that thread w.r.t. the postcondition in which there is no obligation, and (13-b) for any condition variable for which a thread is waiting the invariant $\mathsf{enoughObs}$ holds.

▶ **Theorem 31** (Small Steps Preserve Validity of Configurations)**.** *Each step of the execution preserves validity of configurations.*

$$(t,h) \rightsquigarrow (t',h') \wedge \mathsf{valid}_{n+1}(t,h) \Rightarrow \mathsf{valid}_n(t',h')$$

Proof. By case analysis of the small step relation $\rightsquigarrow$. ◀

▶ **Theorem 32** (A Valid Configuration Is Not Deadlocked)**.** *If a valid configuration has some threads then there exists a thread in this configuration neither waiting for a condition variable nor a lock.*

$$\mathsf{valid}_n(t,h) \wedge \exists id.\ t(id) \neq \varnothing \Rightarrow \exists id'.\ \mathsf{waiting\_for}(\mathsf{fst}(t(id'))) = \varnothing$$

Proof. We assume that all threads in t are waiting for a condition variable or a lock. Since (t,h) is a valid configuration there exists a valid list of augmented threads A with a corresponding valid bag $G = \mathsf{valid\_bag}(A)$, where $\mathsf{valid\_bag}$ maps any element (id,c,ξ,p,O,g) to an element $(\mathsf{waiting\_for}(c), O)$. By Lemma 33, we have $G{=}\{\!|\,|\!\}$, implying $A{=}\{\}$, implying $t{=}\mathbf{0}$ which contradicts the hypothesis of the theorem.

Note that in the definition of validity of a configuration we also keep track of all locations whose addresses are allocated, which makes it possible to provide the function R, mapping lock and condition variable addresses to their levels, for Lemma 33. The first hypothesis in this lemma is met by the constraint 13-a in the definition of validity of a configuration. Additionally, the second hypothesis in this lemma is met by the constraints 12 or 13-b, where the related thread is waiting for a lock or a condition variable, respectively. ◀

$$\mathsf{ol}(m,r) ::= (m.l, r, m.l)$$
$$\mathsf{ov}(m,r) ::= (m.v, r, m.l)$$
$$\mathsf{l}(m,r) ::= (\mathsf{ol}(m,r), (\mathsf{linv}, [m]), (0,[]), \{\!\}\!)$$
$$\mathsf{v}(m,r) ::= (\mathsf{ov}(m,r), (0,[]), (\mathsf{M},[]), \{\!|\mathsf{ov}(m,r)|\!\})$$

$$\mathsf{mutex}(\mathsf{mutex}\ m, \mathsf{waitobj}\ o) = \mathsf{lock}(\mathsf{l}(m, \mathsf{R}(o){-}1)) * \mathsf{cond}(o) \wedge o = \mathsf{v}(m, \mathsf{R}(o))$$

$$\mathsf{pt}(\mathsf{linv}, [m]{\cdot}args) = \lambda\, Wt.\ \lambda Ot.\ \exists b, w.\ m.b \mapsto b * m.w \mapsto w \wedge\ Wt(m.v){=}w \wedge$$
$$(0 < b\ ?\ 0 < Ot(v) : Wt(v) = 0)$$
$$\mathsf{pt}(\mathsf{M}, args) = \lambda\, Wt.\ \lambda Ot.\ \mathsf{true}$$

Figure 44 Verification of the fair mutexes implementation shown in Figure 16 using the proof rules in Figure 43 (part one of two).

Lemma 33 ensures that in any state of the execution if all the desired invariants are respected then it is impossible that all threads are waiting for an object. In this lemma G is a bag of object-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 associated with the object for which t is waiting and the second element is associated with the obligations of t.

▶ **Lemma 33** (A Valid Bag of Augmented Threads Is Not Deadlocked).

$$\forall\, G : Bags(Addresses \times Bags(Addresses)), R : Addresses{\to}Levels.$$
$$(\forall (o,O){\in}G.\ o \prec O \wedge (\exists o', O'.\ (o', \{\!|o|\!\}{\uplus}O') \in G)) \Rightarrow G = \{\!\}$$

Proof. By contradiction; assume that $\exists (o_m, O_1) \in G$ where $\neg\exists (o,O) \in G.\ R(o) < R(o_m)$. By H_2 we have $\exists o', O'.\ (o', \{\!|o_m|\!\}{\uplus}O') \in G$ and by H_1 we have $o' \prec \{\!|o_m|\!\}{\uplus}O'$, which contradicts minimality of the level of o_m. ◀

▶ **Theorem 34** (The Initial Configuration Is Valid). *The initial configuration, consisting of an empty heap and a single thread whose program is verified by the proposed proof rules, is a valid configuration.*

$$\mathsf{correct}_{sp}(\mathsf{obs}(\{\!\}), c, \lambda\_.\mathsf{obs}(\{\!\})) \Rightarrow \forall n, id.\ \mathsf{valid}_n(\mathbf{0}[id{:=}c; \mathsf{done}], \mathbf{0})$$

Proof. The goal is achieved because there are an augmented thread list $T{=}[(id, c, \mathsf{done}, \mathbf{0}, \{\!\}, \mathbf{0})]$, a list of lock-invariant pairs $\mathsf{Linv}{=}[]$, two permission heaps $p_i{=}\mathbf{0}$ and $p_l{=}\mathbf{0}$, and two ghost resource heaps $g_i{=}\mathbf{0}$ and $g_l{=}\mathbf{0}$, such that all the conditions in the definition of validity of configurations are met. ◀

D.6 An Example Proof

In this section we show how the program in Figure 16 can be verified using the proof rules in Figure 43, as shown in Figures 44 and 45[16].

[16] Note that for this program we assume a straightforward extension of the programming language with immutable structures, i.e. tuples with named components.

routine new_mutex(){
req : {true}
l := new_lock;
{ulock(((l, r−1, l), (0, []), (0, []), {}), {}, {})}
v := new_cvar; nop; //Rule InitCV
{ulock(((l, r−1, l), (0, []), (0, []), {}), {}, {}) ∗ cond((v, r, l), (0, []), (M, []), {(v, r, l)})}
m := mutex(l:=l, v:=v, b:=new_int(1), w:=new_int(1)); nop; //Rule InitLock
m **ens** : {λm. ∃o. mutex(m, o) ∧ R(o)=r}}

routine enter_cs(mutex m){
req : {obs(O) ∗ mutex(m, o) ∧ O(o) ≺ O}
acquire(m.l);
// Let l = l(m, R(o)−1) and ol = ol(m, R(o)−1)
{obs(O⊎{ol}) ∗ mutex(m, o) ∗ ∃ Wt, Ot. locked(l, Wt, Ot) ∗ pt(linv, [m])(Wt, Ot)}
if(m.b){
m.w := m.w+1;
{obs(O⊎{ol}) ∗ mutex(m, o) ∗ ∃ Wt, Ot. locked(l, Wt, Ot) ∗ pt(linv, [m])(Wt⊎{m.v}, Ot)}
wait(m.v, m.l)
{obs(O⊎{ol, O(o)}) ∗ mutex(m, o) ∗ ∃ Wt, Ot. locked(l, Wt, Ot) ∗ pt(linv, [m])(Wt, Ot)}}
else{ m.b := 1; nop; //Rule ChargeObligation
{obs(O⊎{ol, O(o)}) ∗ mutex(m, o) ∗ ∃ Wt, Ot. locked(l, Wt, Ot) ∗ pt(linv, [m])(Wt, Ot)}};
{obs(O⊎{ol, O(o)}) ∗ mutex(m, o) ∗ ∃ Wt, Ot. locked(l, Wt, Ot) ∗ pt(linv, [m])(Wt, Ot)}
release(m.l)
ens : {obs(O⊎{O(o)}) ∗ mutex(m, o)}}

routine exit_cs(mutex m){
req : {obs(O⊎{O(o)}) ∗ mutex(m, o) ∧ O(o) ≺ O}
acquire(m.l);
// Let l = l(m, R(o)−1) and ol = ol(m, R(o)−1)
{obs(O⊎{ol, O(o)}) ∗ mutex(m, o) ∗ ∃ Wt, Ot. locked(l, Wt, Ot) ∗ pt(linv, [m])(Wt, Ot)}
m.b := 0;
if(0<m.w){ m.w := m.w−1; m.b := 1; notify(m.v)
{obs(O⊎{ol}) ∗ mutex(m, o) ∗ ∃ Wt, Ot. locked(l, Wt, Ot) ∗ pt(linv, [m])(Wt, Ot)}}
else{ nop //Rule DischargeObligation
{obs(O⊎{ol}) ∗ mutex(m, o) ∗ ∃ Wt, Ot. locked(l, Wt, Ot) ∗ pt(linv, [m])(Wt, Ot)}};
release(m.l)
ens : {obs(O) ∗ mutex(m, o)}}

Figure 45 Verification of the fair mutexes implementation shown in Figure 16 using the proof rules in Figure 43 (part two of two).