Deadlock-free Channels and Locks

K. Rustan M. Leino [0], Peter Müller [1], and Jan Smans [2]

[0] Microsoft Research, Redmond, WA, USA, leino@microsoft.com
[1] ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch
[2] KU Leuven, Belgium, jan.smans@cs.kuleuven.be

Abstract. The combination of message passing and locking to protect shared state is a useful concurrency pattern. However, programs that employ this pattern are susceptible to deadlock. That is, the execution may reach a state where each thread in a set waits for another thread in that set to release a lock or send a message.
This paper proposes a modular verification technique that prevents deadlocks in programs that use both message passing and locking. The approach prevents deadlocks by enforcing two rules: (0) a blocking receive is allowed only if another thread holds an obligation to send and (1) each thread must perform acquire and receive operations in accordance with a global order. The approach is proven sound and has been implemented in the Chalice program verifier.

0 Introduction

Concurrent threads of execution communicate and synchronize using various paradigms. One paradigm is to let threads have shared access to certain memory locations, but to insist that each thread accesses the shared memory only when holding a mutual-exclusion lock. Two familiar programming errors that can occur with this paradigm are forgetting to acquire a lock when accessing shared state and *deadlocks*, that is, not preventing situations where in a set of threads each is waiting to acquire a lock that some other thread in the set is currently holding. Another paradigm is to let threads synchronize by sending and receiving messages along channels. In a pure setting with channels, there are no shared memory locations and data is instead included in the messages. Deadlocks are possible programming errors in this setting, too. Here, a deadlock occurs when a set of threads each is waiting to receive a message from another thread in the set.

Because each of these two paradigms is especially natural for solving certain kinds of problems, there are also situations where one wants to use a combination of the paradigms. For example, consider a concurrently accessed binary tree protected by mutual-exclusion locks. An iterator of this data structure uses locks to read elements from the tree, but may choose to provide these elements to clients via channels, which are more suitable for that task. In the combined setting, a deadlock occurs when a set of threads each waits for another thread in that set to release a lock or send a message.

In this paper, we consider program verification in the combined setting. In particular, we present a technique for specifying programs in such a way that they can be verified to be free of deadlock. Our technique is *modular*, meaning that the verifier can be run on each part of a program separately. We consider multiple-writer, multiple-reader, copy-free channels with infinite slack, that is, with non-blocking sends. The channels are first

class, meaning they can themselves be stored as shared data or passed along channels. We describe the work in the context of the prototype language and verifier Chalice.

This paper is structured as follows. Sec. 1 describes the existing features of the Chalice program verifier that are relevant to this paper. In Sec. 2 and 3, we extend Chalice with channels and show how deadlock can be avoided. The formal details of the verification technique together with a soundness proof are then given in Sec. 4 and 5.

1 Background on Chalice

Chalice [21, 22] is a programming language and program verifier for concurrent programming. The language supports dynamically allocated objects and allows programs to include *contracts* (specifications). The verifier detects common bugs such as data races, null dereferences, violations of assertions and other contracts, and deadlocks. If a program passes the verifier, it is compiled (via C#) to executable code for the .NET platform. The executable code is free of contracts and ghost state, which the verifier confirmed to hold and which were used only to make the verification go through. In this section, we highlight Chalice's features that are relevant to this paper: permissions, locks, and deadlock prevention; see [22] for a full tutorial.

1.0 Permissions

Verification in Chalice centers around permissions and permission transfer. Conceptually, each activation record holds a set of permissions. A memory location can be read or written by an activation record only if it has permission to do so. In this paper, we do not distinguish between read and write permissions, but see [21]. We denote the permission to access the field `f` of an object `o` by **`acc`**`(o.f)`. Our implementation provides *predicates* to abstract over permissions and to express permissions of whole object structures [22], but we omit them here for simplicity. Permissions are part of the ghost state used to reason about programs, but they are not represented in executable code.

The set of permissions held by an activation record can change over time. More specifically, when a new object is created, the creating activation record gains access to the fields of the new object. For example, when the method `Main` of Fig. 0 creates the object `a`, it gets permission to access `a.balance`, and thus it is allowed to set `a.balance` to `10` on the next line. In a similar fashion, `Main` receives permission to access `b.balance` when creating `b`. The fourth statement of `Main` is a method call: **`call`** `b.SetBalance(20);`. Execution of a method call starts by pushing a new activation record onto the stack. What permissions does this new activation record initially have? The answer to this question is determined by looking at the precondition (keyword **`requires`**) of `SetBalance`, which indicates that the caller must hold the permission to access **`this`**`.balance`. This permission transfers from the caller to the callee on entry to the method. In a similar fashion, the postcondition (keyword **`ensures`**) indicates what permissions transfer from the callee to the caller when the method returns. In our example, `Main` gives away its permission to access `b.balance` when it calls `b.SetBalance(20)`. The activation record `b.SetBalance(20)` uses this permission to justify its update of `b.balance`, and then passes the permission back to the caller. That

is, SetBalance effectively just borrows the permission from Main; in general, however, a method need not always return the permissions stipulated by its precondition.

```
class Account {
  var balance: int;

  invariant acc(this.balance);

  method SetBalance(a: int)
    requires acc(balance);
    ensures acc(balance) && balance == a;
  { balance := a; }

  method Transfer(from: Account, to: Account, amount: int)
    requires waitlevel << from.mu && from.mu << to.mu;
  {
    acquire from;
    acquire to;
    fork tok := to.SetBalance(to.balance + amount);
    call from.SetBalance(from.balance - amount);
    join tok;
    release to;
    release from;
  }

  method Main()
  {
    var a := new Account;
    a.balance := 10;
    var b := new Account;
    call b.SetBalance(20);
    share a above waitlevel; share b above a;
    call Transfer(a, b, 5);
  }
}
```

Fig. 0. A small Chalice program illustrating permissions, permission transfer, locks, and deadlock prevention.

If an activation record does not return the permissions that it may still hold at the end of the method, then those permissions are lost forever. In effect, this renders some fields inaccessible. We say that the method *leaks* the permissions, which is allowed.[0]

In addition to calls, Chalice supports **fork** statements. Just like an ordinary call, execution of a **fork** statement leads to the creation of a new activation record. However,

[0] The Chalice verifier has a -checkLeaks option that verifies the absence of leaking. An unused object can then be returned to the system, along with the permissions to its fields.

the new activation record is not pushed onto the current stack, but rather a new thread with its own stack is created and the callee is executed by the new thread. A **fork** operation is non-blocking. That is, the forking thread does not wait for the forkee to run to completion; instead, the forking and forked threads execute concurrently. Using a **join** statement, one thread can wait for another to complete. More specifically, **fork** returns a token, and a **join** on a token causes the joining thread to wait for the completion of the thread corresponding to the token. A token is allowed to be joined only once. Similarly to an ordinary call statement, the activation record that does the **fork** loses the permissions entailed by the precondition of the forkee, and the activation record that completes the corresponding **join** gains the permissions entailed by the postcondition. In our example, the forked activation record for `SetBalance` (in the method `Transfer`) obtains access permission to `from.balance`, and this permission is returned at the join statement.

Note that each **call** statement can be considered to be syntactic sugar for a **fork** statement immediately followed by a corresponding **join**.

Chalice enforces that when one thread holds full permission to a memory location, then no other thread can hold any permission to that memory location. This prevents race conditions and lets the verifier reason about data invariants in the presence of multiple threads.

1.1 Locks

The machinery introduced so far allows synchronization and permission transfer between threads only when threads are forked or joined. However, access to shared data such as a shared buffer requires various threads to obtain and relinquish permissions while the threads are running. Access to shared data can be synchronized using mutual-exclusion *locks*. In Chalice, a lock can hold access permissions, just like an activation record can. Therefore, a thread can pass permissions to another thread by first transferring them to a lock, which allows the other thread to obtain them from the lock.

An object in Chalice can be in one of three states: *not-a-lock*, *available*, and *held*. The object transitions between these states upon execution of a **share**, **acquire**, or **release** statement. A newly allocated object starts in the *not-a-lock* state, where it cannot be used as a lock. The **share** statement initializes a *not-a-lock* object as a lock and transitions the object to the *available* state. The **acquire** operation waits until the object is in the *available* state and then transitions it to the *held* state. The **release** operation transitions the object back to *available*.

A class can declare a *lock invariant* (keyword **invariant**), which indicates, for each lock corresponding to an instance of the class, what permissions are held by the lock when the lock is in the *available* state. For example, the **invariant** declaration in class `Account` of Fig. 0 indicates that the lock corresponding to an `Account` object `o` holds permission to the field `o.balance`. In other words, the lock `o` protects `o.balance`. When an activation record puts an object into the *available* state (by a **share** or **release** operation), it transfers the permissions entailed by the object's invariant to the lock. Conversely, the permissions held by the lock are transferred to an activation record when it completes an **acquire** operation on the lock.

So, when a thread wants to access a shared memory location, it uses an **acquire** operation to compete for the lock that protects the location. Upon successful acquisition of the lock, the permissions held by the lock are transferred to the acquiring thread. When the thread is done accessing the location, it uses the **release** operation to release the lock and transfer the permissions back into the lock. For example, the method `Transfer` in Fig. 0 locks the shared `Account` objects `from` and `to` to gain access to their `balance` fields.

Note that it is the mechanism of permissions that prevents data races. Lock acquisition is one way to obtain permissions, but the act of holding a lock does not by itself imply any rights to access memory.

1.2 Deadlock Prevention

To ensure mutual exclusion, the **acquire** operation suspends the execution of the acquiring thread until the requested lock can be given to that thread. A well-behaved program makes sure that other threads will eventually make such a lock available.

Chalice prevents deadlocks by breaking cycles among acquiring threads. This is done by letting a program associate each lock with a *wait level* and then checking that the program acquires the locks in strict ascending order. The wait levels are values from a set Mu, which is a dense partial order with a bottom element. Chalice uses << to denote the strict partial order on Mu. A program specifies the wait level of a lock using the **share** statement, which takes an optional **between ... and ...** clause. Alternatively, a clause **above ...** or **below ...** may be used if only one bound is given. By default, the **share** statement uses **above waitlevel**, where **waitlevel** denotes the highest lock currently held by the thread. For example, method `Main` in Fig. 0 shares `a` and `b` to make them available for locking. Since `b` is shared above `a`, a thread that holds `b` is not allowed to acquire `a`.

The wait level of an object is recorded in a ghost field called `mu`. In this paper, we assume `mu` to be immutable, that is, once a lock has been shared with a certain wait level, that level cannot change. Our previous work [21] permits dynamic lock re-ordering, which we omit here to focus on the essentials. Since `mu` is immutable, accesses to `mu` do not require any permissions. In Fig. 0, `Transfer`'s precondition demands (0) that the current thread only hold locks whose wait level is strictly below `from.mu` and (1) that `from`'s level lie below `to`'s level.

2 Channels

A channel is an unbounded message buffer with two operations, **send** and **receive**. The former operation adds a message to the buffer, while the latter blocks until a message becomes available, removes that message from the buffer, and returns it to the receiving thread. A channel may declare a *message invariant* (keyword **where**), which constrains the messages sent over the channel and also specifies permissions that are transferred over the channel along with each message.

As an example, consider the program of Fig. 1. The first two lines declare a new channel type `Ch` with two parameters `p0` and `p1`. These parameters indicate that each

message for a Ch channel object consists of two Person objects. The **where** clause states that each message in a Ch channel carries the permissions for accessing the age field of the persons passed as parameters. In addition, it specifies that p0 must be at least 18 years old. The method Main creates two persons, cooper and dylan, sends a message on the channel ch, and finally receives a message on that channel. When an activation record sends a message, the permissions entailed by the **where** clause transfer from the sender to the message. Similarly, when a message is received, the permissions in the message transfer to the receiving activation record. The mechanism makes the channels *copy-less*, because only the object references among the message parameters, not the data fields accessed from those references, are sent over the channel.

```
channel Ch(p0: Person, p1: Person)
  where acc(p0.age) && acc(p1.age) && 18 <= p0.age;

class Person {
  var age: int;

  method Main() {
    var cooper := new Person; var dylan := new Person; cooper.age := 62;
    var ch := new Ch;
    send ch(cooper, dylan);
    // ...
    receive a, b := ch;
  }
}
```

Fig. 1. Declaration of a channel type Ch and a Main method that sends and receives.

Note that, analogous to the semantics of pre- and postconditions and lock invariants, it is an error if at a **send** statement the sender lacks the permissions entailed by the message invariant or if the other constraints in the message invariant are not satisfied. For instance, if we omitted the update cooper.age := 62;, then the verifier would report that the last constraint in the **where** clause does not hold.

A program using channels can deadlock if a thread is blocked on a **receive** statement, waiting for a message that is never sent. For example, consider the following code snippet.

```
ch := new Ch;
receive a, b := ch; // deadlock
```

This program deadlocks, since the thread is blocked forever at the **receive** statement. To avoid such deadlocks, we impose the restriction that a thread may perform a **receive** statement only if there are sufficient messages in the channel or other threads hold obligations to send. We enforce the restriction as follows.

In addition to permissions, each activation record holds a number of credits. We denote the right to receive n messages ($0 \leq$ n) on channel ch by **credit**(ch, n). The obligation to send n messages on ch is denoted by **credit**(ch, -n). We sometimes refer to a negative credit as a debt. **credit** predicates can be used in specifications. Multiple occurrences of a **credit** predicate are equivalent to one predicate with the sum of the credits, that is, **credit**(ch,i) && **credit**(ch,j) is the same as **credit**(ch,i+j).

A **receive** statement is allowed only if the activation record holds at least one credit on the corresponding channel. Execution of a **receive** statement decreases the number of credits by one. Conversely, a **send** statement increases the number of credits by one. However, threads can always send messages without regard to the number of credits. While positive credits (permissions to receive) can be leaked at the end of method bodies, negative credits (obligations to send) must always be returned to the caller. These rules enforce the invariant that the total sum of the number of credits for a channel Ch never exceeds the number of items stored in the channel.

Just like permissions can be transferred between activation records (specified by **requires** and **ensures**), so can credits. For example, in the program in Fig. 2, the Main method transfers a debt to Producer. That is, Main decreases its balance for ch by -1, resulting in a positive balance for Main. Consequently, Producer starts with an obligation to send and Main has obtained permission to receive. Main then transfers a credit to Consumer, resulting in a 0 credit balance for Main.

Also, just like permissions can be stored in lock invariants and message invariants, so can *positive* credits. For example, every message with a non-negative x parameter in Fig. 2 entails a credit. Thus, in effect, Producer puts into each such message a promise that it will send yet another message, and this credit sent along the channel allows Consumer to "pay" for its next **receive** operation.

Storing negative credits in lock or message invariants is not allowed. Since a program need not acquire all available locks or receive all sent messages eventually, allowing negative credits here would be a way to hide debt. We enforce this requirement by a simple proof obligation for each lock and message invariant. Moreover, a **call** is allowed only if transferring the credits entailed by the precondition does not bring the caller into debt. This requirement is necessary to prevent a thread from creating a credit by a simple local method call. The callee could use the credit to receive, but the caller, which has the obligation to send and which executes in the same thread, would never continue its execution, and the program deadlocks. This restriction does not apply to **fork**, because there the forker will continue its execution and, thus, can live up to is obligation to send.

The credit accounting introduced so far handles channels with blocking receives and non-blocking sends. We can also support channels with finite slack (that is, blocking sends) by distinguishing between the receive credits described above and send credits. We omit a discussion of this extension because it does not reveal anything interesting.

In many languages, channels can be implemented using locks and condition variables. Channels have the advantage that each send operation earns a credit because it puts a message in the buffer. In contrast, a signal operation on a condition variable is lost when no thread is currently waiting on the condition variable. Therefore, one cannot

```
channel Ch(x: int) where 0 <= x ==> credit(this, 1);

class ProducerConsumer {
  method Produce(ch: Ch)
    requires credit(ch, -1);
  {
    var i := 0;
    while(i < 10)
      invariant 0 <= i && i <= 10 && credit(ch, -1);
    { send ch(i); i := i + 1; }

    send ch(-1);
  }

  method Consume(ch: Ch)
    requires credit(ch, 1) && waitlevel << ch.mu;
  {
    var x: int;
    receive x := ch;
    while(0 <= x)
      invariant waitlevel << ch.mu;
      invariant 0 <= x ==> credit(ch, 1);
    { receive x := ch; }
  }

  method Main() {
    var ch := new Ch;
    fork Produce(ch);
    fork Consume(ch);
  }
}
```

Fig. 2. Producer/Consumer example illustrating the use of channels. Operator ==> denotes short-circuit boolean implication. The loop invariant (keyword **invariant**) specifies what is given to each new iteration and what must be returned by each completed iteration.

decide locally whether a signal operation earned a credit or not. This difference makes it much harder to prove deadlock freedom for condition variables than for channels.

3 Global Wait Order

The rules described in the previous section enforce the invariant that, for each receiving thread, either the corresponding channel contains a message or a thread holds the obligation to send. However, this invariant does not suffice to rule out deadlocks. A deadlock can still occur if execution reaches a state where a subset of the running threads is waiting for another thread in that set to send a message.

As an example, consider the program of Fig. 3. Both the main thread and the forkee block at their respective **receive** statements and wait forever for the other to send. A similar situation can occur when combining locks and channels. For example, the main thread in Fig. 4 waits for a message on channel ch, while the forkee waits for the main thread to release the lock. Note that both of these programs satisfy the rules described in the previous section. In particular, at each **receive** statement, the credits held by the activation record on the corresponding channel are strictly positive and no debt is leaked at the end of methods.

In the combined setting with locks and channels, we say a deadlock occurs if each of a set of threads is waiting for another thread in that set to either send a message or release a lock (or formally, if the graph corresponding to a configuration as defined in Definition 2 contains a cycle). We break cycles and prevent deadlocks in the combined setting by using a global wait order that includes locks and channels. Just as locks, channels have a wait level that is stored in the ghost field mu. For channels, the ghost field mu is set (using a **between** clause) when the channel is created. Receiving on a channel ch requires ch.mu to be larger than **waitlevel**. We redefine **waitlevel** as the larger of: the largest object whose lock is held by the thread and the largest channel for which the thread has an obligation to send.

```
channel Ch() where true;

class Program {
  method M(ch0: Ch, ch1: Ch)
    requires ch0 != ch1;
    requires credit(ch0, 1) && credit(ch1, -1);
  { receive ch0; send ch1(); }

  method Main() {
    var ch0 := new Ch; var ch1 := new Ch;
    fork M(ch0, ch1);
    receive ch1; send ch0();
  }
}
```

Fig. 3. A program that deadlocks using just channels.

```
channel Ch() where true;

class Program {
  method M(ch: Ch)
    requires credit(ch, -1);
  { acquire this; send ch(); release this; }

  method Main() {
    var ch := new Ch;
    acquire this;
    fork M(ch);
    receive ch;
    release this;
  }
}
```

Fig. 4. A program that deadlocks using channels and locks.

The additional restrictions outlined above cause verification of the programs in Figs. 3 and 4 to fail. The first program does not verify (and cannot be made to verify by adding further specifications) because `ch0.mu` and `ch1.mu` cannot both be larger than the other. This means that either the **`receive`** statement in the main thread or in the forkee is disallowed, as the wait level of the corresponding channel does not lie above **`waitlevel`** of the respective thread. The second program does not verify because either the lock in the acquire statement in `M` or the channel of the **`receive`** statement in the main thread does not lie above **`waitlevel`** of the respective thread.

Besides **`acquire`** and **`receive`**, **`join`** is the third Chalice statement that might cause a thread to wait and is, thus, relevant for deadlock prevention. For instance, a thread might wait to receive a message before terminating while another thread joins the first thread before sending the awaited message. In this paper, we encode **`join`** statements via channels: Each method receives an extra parameter, a channel, and an obligation to send one message on that channel. Before forking, the forker must create a new channel above its wait level and pass it to the forkee. The forkee must send a message on that channel right before it terminates. A thread can then join another thread by receiving on the designated channel. The obligation to send on the designated channel increases the wait level of the forkee above the wait level of the forker, which prevents cyclic waiting. In this encoding, a **`call`** statement is encoded by a **`fork`** immediately followed by a **`join`**. This encoding simplifies the presentation of the proof rules and the soundness argument; programs may still contain **`call`** and **`join`** statements.

4 Verification

In this section, we make the informal rules described in previous sections precise. We define the proof rules for the most interesting statements by translating them into a pseudo-code language, whose weakest precondition semantics is obvious. In this trans-

lation, we use **assert** statements to denote proof obligations and **assume** statements to state assumptions that can be used to prove the assertions. We encode the heap as a two-dimensional array that maps object references and field names to values. The current heap is denoted by the global variable $Heap$. To avoid clutter, we omit null reference checking from the formalization. A program verifier can be built from these rules by writing the pseudo code in an intermediate verification language like Boogie [2]. In fact, the pseudo code we use is essentially Boogie 2, and this is how we implemented the Chalice verifier.

4.0 Encoding of Permissions and Credits

Conceptually, each activation record holds a number of permissions and credits. We track permissions during verification via a global variable $\mathcal{P}$. $\mathcal{P}$ is a two-dimensional map from object references and field names to permissions. For simplicity, we encode permissions as boolean values in this paper, but the Chalice verifier supports fractional permissions [21]. An activation record has access to `o.f` if and only if $\mathcal{P}[o, f]$ equals **true**.

In a similar fashion, we track credits via a global variable $\mathcal{C}$, a map from channel instances to integers. $\mathcal{C}[o]$ denotes the number of credits held by the current activation record for the channel o.

4.1 Encoding of Locks and Wait Levels

Our encoding introduces a thread-local variable λ, which yields the set of all objects whose locks are held by the thread of the current activation record.

The Chalice expression **waitlevel** is then encoded as the maximum of the wait levels of locks held by the current activation record and of channels for which the current activation record has an obligation to send:

$$\textbf{waitlevel} \quad \equiv \quad max(\{\ o.mu \mid o \in \lambda\ \} \cup \{\ c.mu \mid \mathcal{C}[c] < 0\ \})$$

For convenience, we will use **waitlevel** in the pseudo code below as an abbreviation for this encoding.

4.2 Encoding of Permission and Credit Transfer

In Chalice, permissions and credits often transfer from and to activation records. For each statement, the set of permissions and credits being transferred is described by an assertion. For example, when a message is sent, the permissions and credits described by the channel's **where** clause transfer from the activation record to the message. Similarly, when a lock is acquired, the permissions and credits described by the lock invariant transfer from the lock to the acquiring activation record. In our verification, we model permission and credit transfer via two operations, Inhale and Exhale. The former operation adds the permissions and credits described by an assertion to the activation record's $\mathcal{P}$ and $\mathcal{C}$, while the latter removes them.

$\mathsf{Inhale}[\![\texttt{acc(o.f)}]\!] \equiv$
  $\mathbf{havoc}\ \mathit{Heap}[o, f]$;
  $\mathcal{P}[o, f]{:=}\mathbf{true}$;

$\mathsf{Exhale}[\![\texttt{acc(o.f)}]\!] \equiv$
  $\mathbf{assert}\ \mathcal{P}[o, f] = \mathbf{true}$;
  $\mathcal{P}[o, f]{:=}\mathbf{false}$;

$\mathsf{Inhale}[\![\texttt{credit(ch, n)}]\!] \equiv$
  $\mathcal{C}[ch]{:=}\mathcal{C}[ch] + n$;

$\mathsf{Exhale}[\![\texttt{credit(ch, n)}]\!] \equiv$
  $\mathcal{C}[ch]{:=}\mathcal{C}[ch] - n$;

$\mathsf{Inhale}[\![P\ \texttt{\&\&}\ Q]\!] \equiv$
  $\mathsf{Inhale}[\![P]\!]$;
  $\mathsf{Inhale}[\![Q]\!]$;

$\mathsf{Exhale}[\![P\ \texttt{\&\&}\ Q]\!] \equiv$
  $\mathsf{Exhale}[\![P]\!]$;
  $\mathsf{Exhale}[\![Q]\!]$;

Otherwise :
$\mathsf{Inhale}[\![E]\!] \equiv$
  $\mathbf{assume}\ E$;

Otherwise :
$\mathsf{Exhale}[\![E]\!] \equiv$
  $\mathbf{assert}\ E$;

Fig. 5. Transfer of permissions and credits via Inhale and Exhale.

The definitions for Inhale and Exhale are shown in Fig. 5. When an activation record obtains permission to access `o.f` by inhaling the permission, we assign an arbitrary value to $\mathit{Heap}[o.f]$ (keyword **havoc**) since other threads may have updated the location while it was not accessible to the current thread. Inhaling **`credit`**`(ch, n)` increases the number of credits for ch by n. Inhaling a conjunction P `&&` Q corresponds to first inhaling P and afterwards inhaling Q. If the inhaled assertion is a pure boolean expression (that is, contains neither access nor credit predicates), we assume the expression holds.

Exhaling permission to access `o.f` corresponds to removing the permission. However, exhaling permissions is allowed only if the permission is present. Exhaling credits corresponds to decrementing the credit map. Exhaling a conjunction P `&&` Q corresponds to first exhaling P and afterwards exhaling Q. Finally, exhaling a pure assertion corresponds to proving that the assertion holds.

4.3 Encoding of Channel Operations

Channels support three operations: creation, sending, and receiving. The translation to pseudo code for each of these statements is shown in Fig. 6.

A channel creation `x := new Ch between l and u;` creates a new channel whose `mu` field lies between l and u.[1] To guarantee a wait level exists that lies between l and u, we first check that l is strictly smaller than u. Then, we assign an arbitrary channel identifier to x, such that the current thread has no credits for that channel x and such that $l << x.mu << u$.

The statement **`send`** `ch(`$\texttt{x}_1$`, ..., `$\texttt{x}_n$`);` adds a new message to the channel ch and earns a credit, which is reflected in the credit map of the sending activation record. The permissions and credits described by the **`where`** clause transfer from the activation record to the message (encoded by Exhale).

[1] For simplicity, we consider only a single lower and upper bound. However, our implementation supports an arbitrary number of bounds.

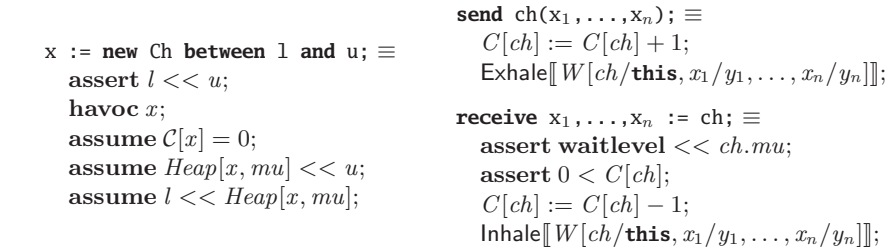

Fig. 6. Translation to pseudo code for channel operations. For **new**, we omitted some details that encode that the new channel is different from all previously existing channels. For **send** and **receive**, ch is assumed to have type **channel** Ch(y_1: t_1, ..., y_n: t_n) **where** W.

The statement **receive** x_1, ..., x_n := ch; removes a message from the channel ch. Receiving is allowed only if **waitlevel** is smaller than the wait level of ch and if the current activation record holds at least one credit for ch. Since receiving removes a message from the channel, the number of credits is decremented by one. The permissions and credits described by the **where** clause transfer from the message to the receiving activation record.

4.4 Encoding of Fork, Join, and Call

As described at the end of Sec. 3, we encode **join** statements via channels, and **call** statements via **fork** and **join**. In this encoding, we make the following modifications for each method m(p: T) **returns** (r: R) with precondition P and postcondition Q in a class C: (0) We declare a channel type Ch_m(t: C, p: T, r: R) **where** Q';. Q' is Q with t substituted for **this**. (1) We add a parameter j: Ch_m to m. (2) We add a precondition **credit**(j, -1) to m, which expresses that the method has an obligation to send a message on the join-channel j. (3) We add a precondition j.mu << u_i.mu for each channel expression u_i that occurs in a credit expression in m's precondition P. This precondition allows m to receive on the channels u_i even though it has a debt for channel j. (4) At the end of m's body, we place a send statement **send** j(**this**, p, r);. This send lives up to the obligation expressed by the precondition (2). (5) We remove the postcondition Q from m because all information, permissions, and credits are conveyed to the caller via the send operation (4).

We encode **fork** tok := x.m(y) as **var** tok := **new** Ch_m **above waitlevel** **below** u_1, ..., u_n; **fork** x.m(y, tok);. That is, a **fork** passes a new channel instance to the forkee and transfers the permissions and credits described by the forkee's precondition P. The wait level of the join-channel lies above **waitlevel**, but below each channel u_i that occurs in a credit expression in m's precondition. This allows the forker to join the forkee and it allows the forkee to perform receive statements on the channels u_i. To allow the forkee to acquire locks, one also has to ensure that the wait level of the join-channel tok is below each lock that the forkee might want to acquire. Choosing such a level is possible, but we omit the details for simplicity.

We encode `join z := tok;` by receiving on the channel we passed to the thread when it was forked: `var t: C; var p: T; receive t,p,z := tok;`. This receive inhales the message invariant of $\texttt{Ch}_m$ (that is, the joined thread's postcondition) and transfers permissions and credits accordingly.

We translate a call `call z := x.m(y);` into a fork immediately followed by a join: `fork tok := x.m(y); join z := tok;`, which are then further encoded as described above. This encoding automatically satisfies the rule that a call must not create a debt in the caller (see Sec. 2).

5 Deadlock Freedom

In this section, we prove that the verification technique described in the previous sections indeed prevents deadlocks. However, we provide only the key definitions and lemmas. For the full proof, we refer the reader to [23].

Note that our verification technique proves partial correctness, that is, it considers non-terminating methods to be correct. As a consequence, we do not prevent situations where a thread waits forever on another thread to send a message or release a lock, and that other thread ran into an infinite loop or recursion before executing the awaited operation. Proving termination of loops and recursion is an orthogonal issue.

5.0 Language

For the proof of deadlock freedom, we use a smaller programming language that omits all features that are not relevant for the proof such as permissions, classes, and the heap.

We prove soundness with respect to the language of Fig. 7. A program consists of a number of declarations and a main routine $\overline{s}$. A declaration is either a channel or a procedure. Each channel has a channel name, channel parameters, and a **where** clause. Each procedure has a procedure name, procedure parameters, a precondition, and a body. A statement is an object creation, a send or receive operation, a fork statement, an acquire statement, or a release statement. Finally, an assertion is either credit (of $+1$) or debt (of -1). Note that we do not distinguish objects and channels, and we use both terms interchangeably in the soundness proof.

$$
\begin{array}{lll}
\mathit{program} & ::= & \overline{\mathit{decl}}\ \overline{s} \\
\mathit{decl} & ::= & \mathit{channel} \mid \mathit{procedure} \\
\mathit{channel} & ::= & \textbf{channel}\ C(\overline{x})\ \textbf{where}\ \overline{\phi}; \\
\mathit{procedure} & ::= & \textbf{procedure}\ m(\overline{x})\ \textbf{requires}\ \overline{\phi};\ \{\ \overline{s}\ \} \\
s & ::= & x := \textbf{new}\ C;\ \mid \textbf{send}\ x(\overline{x});\ \mid \textbf{receive}\ \overline{x} := x;\ \mid \\
 & & \quad \textbf{fork}\ m(\overline{x});\ \mid \textbf{acquire}\ x;\ \mid \textbf{release}\ x; \\
\phi & ::= & \textbf{credit}(x) \mid \textbf{debt}(x)
\end{array}
$$

Fig. 7. A small language with lockable channels.

$\mathcal{O}$ is the set of object references, Mu the set of wait levels, and $\mathcal{X}$ the set of variables. The set Mu with the binary operator $<<$ forms a dense partial order. L is

a function that maps each object reference to its wait level. We consider only channels whose **where** clause does not contain debt.

5.1 Execution Semantics

Definition 0 shows that threads can be in one of three states: *running*, *done*, and *aborted*. The job of the program verifier is to ensure that threads do not perform illegal operations and hence that no thread ends up in the *aborted* state. We say that a configuration is *aborting* if one or more threads is in the *aborted* state. A configuration is *final* if each thread is in the *done* state.

Definition 0. *A* thread state σ *is one of the following:*

- $\mathbf{run}(\overline{s}, \Gamma)$*, indicating the thread is running with remaining statement* $\overline{s}$ *and environment* Γ*.* Γ *is a partial function from variable names to object references.*
- $\mathbf{done}$*, indicating the thread has completed.*
- $\mathbf{aborted}$*, indicating the thread has performed an illegal operation.*

Definition 1. *A configuration* ψ *is a pair consisting of:*

- Ω*, a partial function from object references to environment lists. Each environment in the list represents a message. Thus,* $\Omega(o)$ *denotes the list of messages inside channel* o*. We say that an object is* allocated *if* $o \in \mathsf{dom}(\Omega)$*.*
- T*, a multiset of threads. Each thread is a triple* $(\sigma, \kappa, \lambda)$*.* σ *is a thread state,* κ *is a function from object references to integers, and* λ *is a set of object references.* $\kappa(o)$ *denotes the number of credits held by the thread and* λ *is the set of objects locked by the thread.*

Execution of programs is defined by the small-step relation $\rightarrow$ shown in Fig. 8. The rules of Fig. 8 contain premises marked dark gray and premises marked light gray. A premise marked dark gray indicates that threads must block and wait for the premise to become true. For example, a receive statement blocks until the corresponding channel contains a message (*i.e.*, $0 < \mathsf{length}(\Omega(o))$). A premise marked light gray that does not hold indicates that the thread has performed an illegal operation and that the thread can transition to the aborted state. For example, a thread trying to execute a receive statement transfers to the aborted state if the number of credits ($\kappa(o)$) is not strictly positive.

As explained earlier, the job of the program verifier is to ensure that threads do not abort. In other words, the verifier must ensure that the premises marked light gray hold. As a consequence, these premises correspond to the assert statements in the pseudo code of Sec. 4.

$\mathsf{def}(C)$ denotes the definition of channel C in the program. Each object has a corresponding type denoted by $\mathsf{typeof}(o)$. $\mathsf{credits}(\overline{\phi}, \Gamma)$ returns a function from object references to integers, where an entry for channel o indicates the credit associated with the assertion $\overline{\phi}$ for o. $f[a \mapsto b]$ denotes an update of the function f at a with b. If $\mathsf{typeof}(o) = C$ and $\mathsf{def}(C) = \mathbf{channel}\ C(\overline{y})\ \mathbf{where}\ \overline{\phi};$, then $\Phi(o)$ denotes $\overline{\phi}$.

$$\frac{\begin{array}{c}\Gamma(x) = o \qquad \forall i \in \{1,\ldots,n\} \bullet \Gamma(x_i) = o_i \\ \mathsf{typeof}(o) = C \qquad \mathsf{def}(C) = \mathbf{channel}\ C(y_1,\ldots,y_n)\ \mathbf{where}\ \overline{\phi}; \\ \Gamma' = [\mathbf{this} \mapsto o, y_1 \mapsto o_1, \ldots, y_n \mapsto o_n] \\ \kappa' = \mathsf{credits}(\overline{\phi}, \Gamma') \qquad \kappa'' = \kappa[o \mapsto \kappa[o]+1] - \kappa' \qquad \Omega' = \Omega[o \mapsto \Omega(o) + \Gamma']\end{array}}{\begin{array}{c}(\Omega, \{(\mathbf{run}(\mathbf{send}\ x(x_1,\ldots,x_n);\ \overline{s}, \Gamma), \kappa, \lambda)\} \cup T) \rightarrow \\ (\Omega', \{(\mathbf{run}(\overline{s}, \Gamma), \kappa'', \lambda)\} \cup T)\end{array}}$$

$$\frac{\begin{array}{c}\Gamma(x) = o \\ \mathsf{typeof}(o) = C \qquad \mathsf{def}(C) = \mathbf{channel}\ C(y_1,\ldots,y_n)\ \mathbf{where}\ \overline{\phi}; \qquad 0 < \kappa(o) \\ \forall q \in \mathsf{dom}(\Omega) \bullet (\kappa(q) < 0 \vee q \in \lambda) \Rightarrow \mathsf{L}(q) << \mathsf{L}(o) \qquad 0 < \mathsf{length}(\Omega(o)) \\ \Gamma' = \mathsf{head}(\Omega(o)) \qquad \kappa' = \mathsf{credits}(\overline{\phi}, \Gamma') \qquad \kappa'' = \kappa[o \mapsto \kappa[o]-1] + \kappa' \\ \Omega' = \Omega[o \mapsto \mathsf{tail}(\Omega)] \qquad \Gamma'' = \Gamma[x_1 \mapsto \Gamma'(y_1), \ldots, x_n \mapsto \Gamma'(y_n)]\end{array}}{\begin{array}{c}(\Omega, \{(\mathbf{run}(\mathbf{receive}\ x_1,\ldots,x_n := x;\ \overline{s}, \Gamma), \kappa, \lambda)\} \cup T) \rightarrow \\ (\Omega', \{(\mathbf{run}(\overline{s}, \Gamma''), \kappa'', \lambda)\} \cup T)\end{array}}$$

Fig. 8. Execution semantics for well-formed programs (see [23] for all rules).

5.2 Properties

The key property we want to prove is Theorem 2: programs written in the language of Fig. 7 do not get stuck. The proof of this theorem relies on two other theorems, 0 and 1. Theorem 0 states that for each allocated channel c, the total number of credits for c (in activation records and messages) is at most the sum of the amount of debt for c and the number of messages inside c.

Theorem 0. *Suppose* $(\Omega, \{(\sigma_1, \kappa_1, \lambda_1), \ldots, (\sigma_n, \kappa_n, \lambda_n))$ *is a configuration reached by an execution of a well-formed program. Then for each channel* $o \in \mathsf{dom}(\Omega)$*, the following holds:*

$$0 \leq \mathsf{length}(\Omega(o)) - ((\Sigma_{i \in \{1,\ldots,n\}} \kappa_i(o)) + (\Sigma_{q \in dom(\Omega), \Gamma \in \Omega(q)} \mathsf{credits}(\Phi(q), \Gamma)(o)))$$

The proof runs by induction on the length of the execution and by case analysis on the step taken.

Each configuration ψ has a corresponding graph whose nodes are the threads in ψ. This graph contains an edge from thread f to t if f is waiting for t to send a message or to release a lock (see Definition 2). A *deadlock* occurs if the graph contains a cycle. Theorem 1 states that an edge in the graph between f and t implies that t's wait level is smaller than f's wait level. It follows from Theorem 1 that configurations reached by executions of well-formed programs are deadlock-free.

Definition 2. *Each configuration* $(\Omega, \{(\sigma_1, \kappa_1, \lambda_1), \ldots, (\sigma_n, \kappa_n, \lambda_n))$ *has a corresponding graph. The nodes in the graph are threads. The graph has an edge from* $(\sigma_f, \kappa_f, \lambda_f)$ *to* $(\sigma_t, \kappa_t, \lambda_t)$ *if one of the following holds:*

- *Thread* f *waits for* t *to send a message, that is,* σ_f *equals* $\mathbf{run}(\mathbf{receive}\ x;\ \overline{s}, \Gamma)$*,* $\kappa_t(\Gamma(x)) < 0$*, and* σ_f *cannot go to the aborted state.*

– *Thread f waits for t to release a lock, that is, σ_f equals $\mathbf{run}(\mathbf{acquire}\ x;\ \overline{s}, \Gamma)$, $\Gamma(x) \in \lambda_t$, and σ_f cannot go to the aborted state.*

Theorem 1. *Suppose the graph corresponding to a configuration in which no thread is aborted contains an edge from $(\sigma_f, \kappa_f, \lambda_f)$ to $(\sigma_t, \kappa_t, \lambda_t)$. Then the following holds:*

$$\mathsf{max}\{\mathsf{L}(o) | \kappa_f(o) < 0 \vee o \in \lambda_f\} << \mathsf{max}\{\mathsf{L}(o) | \kappa_t(o) < 0 \vee o \in \lambda_t\}$$

The proof runs by induction on the length of the execution and by case analysis on the step taken.

Theorem 2. *Suppose ψ is a non-final, non-aborting configuration. Then, ψ is not stuck.*

Proof. It follows from Theorem 1 that the graph contains a non-final thread t that has no outgoing edges. We have to consider three cases. If the first statement of t is not an acquire or a receive, then t can make progress. If t's first statement is a receive statement for channel o, then no other thread holds debt for o (otherwise t would have an outgoing edge). If $\kappa(o) \leqslant 0$, then the thread can make progress by aborting; otherwise, it follows from Theorem 0 that o contains at least one message and therefore that the thread can make progress. Finally, if the first statement is an acquire for object o, then no other thread holds o's lock (otherwise t would have an outgoing edge). Therefore, t can acquire o.

6 Related Work

Hoare's model of Communicating Sequential Processes (CSP) influentially set the style of languages that communicate over channels [13]. Channels in CSP have no slack, that is, they have no buffer capacity. This means that send and receive operations are executed in a synchronized fashion to form a rendezvous. The channels are named entities, not dynamically created values that can be stored in variables or passed along channels.

Newsqueak is a language that features channels and shared global variables [26]. Like CSP, Newsqueak uses zero-slack channels, but the channels are first class and can be passed around like other references to data structures in memory. After the rendezvous of a sender and receiver, the sender gets a chance to compute its message before it is communicated to the receiver. The language has support for atomic increment and decrement operations, but does not include built-in locking primitives.

The programming language Alef [35] and its successor Limbo [28] apply ideas of Newsqueak to larger programming-language designs. Limbo was designed and used for writing applications for the Inferno operating system. The languages include shared global variables, and locks are provided (Alef) or can be built from channels.

A language with channels that has had considerable success is Erlang [1], a functional language (hence, locks are irrelevant) used in a variety of applications.

Language support for mutual-exclusion locks is provided in several languages, including Modula-3, Java, and C#. Such languages may provide channels in a library, like `ConcurrentLinkedQueue` in Java's `java.util.concurrent` library.

The idea of using permissions to avoid data races was first formulated by Boyland [4] and has been adopted by concurrent separation logic [25]. Several researchers extended concurrent separation logic to handle dynamic thread creation [10, 15, 12], rely/guarantee [31, 8], reentrant locking [11], and channels [14, 27]. Our encoding of permissions in a first-order setting was inspired by implicit dynamic frames [29].

Enforcing the absence of deadlocks by checking that threads acquire locks in accordance with a global order is a well-known technique from operating systems and databases, and has been implemented in several verifiers [6, 9, 16, 17] and static analyzers [20, 3]. To the best of our knowledge, the only existing technique that prevents deadlocks in programs that use channels is Kobayashi's type system for the π-calculus [19]. A channel type in this type system consists of a message type and a usage. The usage describes the order of channel operations and associates an obligation and capability level with each of those operations. The notion of credits in our approach is similar to usages, while wait levels are similar to obligation and capability levels. Kobayashi's type system has two advantages with respect to our approach. First, fewer annotations are required as types can be inferred. Secondly, his approach can handle some programs that we cannot, such as encoding locks via channels. However, the type system has only been applied in the context of the π-calculus, while we integrate deadlock prevention into a verification system for an object-oriented language (which Kobayashi considers to be "useful and important" [18]).

Session types [32, 7] are a technique for checking that channels are used in accordance with a predefined protocol. Recently, Villard *et al.* [34] have integrated the ideas from session types into separation logic. However, the focus of [32, 7, 34] lies in checking conformance of the code with the channel contract and in ensuring memory safety, *i.e.*, that a memory location is not accessed after sending the corresponding permission over a channel. We do not specifically address protocol checking (though protocols can be encoded via ghost state), but we do check memory safety and in addition show how to enforce the absence of deadlocks.

Luecke *et al.* [24] and Vetter *et al.* [33] propose run-time deadlock detection algorithms for systems that use message passing. These algorithms may miss certain deadlocks. Moreover, run-time testing cannot guarantee the absence of deadlocks, since not all paths, thread interleavings, and input values can be considered.

Terauchi and Megacz [30] use ideas similar to those proposed in this paper in a static inference of channel buffer bounds. Their analysis uses a capability mapping per thread, a function from channel identifiers to natural numbers similar to our credit map, to track the number of messages that can be sent for each channel. Just as our credit map, the capability mapping is updated at send and receive statements. Another idea shared by both approaches is that capabilities can be transferred via channels. In particular, each channel type includes a capability mapping (similar to our **`where`** annotation) that describes what capabilities transfer along with messages on channels of that type.

The verification approach presented in this paper prevents non-termination caused by deadlock. However, infinite recursion and loops can still lead to non-terminating executions. For example, a thread may fail to acquire a lock because the thread holding the lock is stuck in an infinite loop. Proving termination of loops and recursion is a separate issue, which for instance can be solved using techniques like [5] and [0].

This paper builds on and extends our earlier work on the Chalice verifier [21]. In particular, we extend Chalice with channels, introduce credits, and insert channels into the wait order to prevent deadlocks involving blocking channel operations.

7 Conclusion

The key contribution of this paper is that it shows how to verify the absence of deadlocks in programs that combine channels and locking. In particular, deadlocks are prevented by enforcing two rules: (0) a blocking receive is allowed only if another thread holds the obligation to send and (1) receive and acquire operations must be done in accordance to a global wait order. The verification technique has been proven sound and was implemented in the Chalice program verifier. As future work, we plan to apply the presented methodology to classical concurrency examples and case studies from programs written in languages that support channels such as Scala and Go. Moreover, we are interested in combining our methodology with termination checking to guarantee that every obligation to send will eventually be fulfilled.

Acknowledgements

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy. We would like to thank Bart Jacobs and the anonymous referees for useful comments and feedback.

References

0. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of Java bytecode. In *FMOODS*, 2008.
1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, second edition, 1996.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*. Springer, 2006.
3. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*. ACM, 2002.
4. J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *LNCS*. Springer, 2003.
5. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*. ACM, 2006.
6. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.
7. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, 2006.
8. X. Feng. Local rely-guarantee reasoning. In *POPL*. ACM, 2009.
9. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*. ACM, 2002.

10. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, volume 4807 of *LNCS*. Springer, 2007.
11. C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In *APLAS*, volume 5356 of *LNCS*. Springer, 2008.
12. C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In *AMAST*, volume 5140 of *LNCS*. Springer, 2008.
13. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8), 1978.
14. T. Hoare and P. O'Hearn. Separation logic semantics for communicating processes. *Electronic Notes on Theoretical Comput. Sci.*, 212, 2008.
15. A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, volume 4960 of *LNCS*. Springer, 2008.
16. B. Jacobs. *A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs*. PhD thesis, Katholieke Universiteit Leuven, 2007.
17. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.
18. N. Kobayashi. Type systems for concurrent programs. In *UNU/IIST 10th Anniversary Colloquium*, 2002.
19. N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, 2006.
20. J. A. Korty. Sema: A Lint-like tool for analyzing semaphore usage in a multithreaded UNIX kernel. In *Proceedings of the Winter 1989 USENIX Conference*. USENIX Association, 1989.
21. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, volume 5502 of *LNCS*. Springer, 2009.
22. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *LNCS*. Springer, 2009.
23. K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks (extended version). Technical Report CW573, Department of Computer Science, K.U.Leuven, 2010.
24. G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11), 2002.
25. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Comput. Sci.*, 375(1–3), 2007.
26. R. Pike. Newsqueak: A language for communicating with mice. Computing Science Technical Report 143, AT&T Bell Laboratories, 1989.
27. D. J. Pym and C. M. N. Tofts. A calculus and logic of resources and processes. *Formal Aspects of Computing*, 18(4), 2006.
28. D. M. Ritchie. The Limbo programming language. In *Inferno Programmer's Manual, Volume 2*. Vita Nuova Holdings Ltd., 2000.
29. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 5653 of *LNCS*. Springer, 2009.
30. T. Terauchi and A. Megacz. Inferring channel buffer bounds via linear programming. In *ESOP*, volume 4960 of *LNCS*. Springer, 2008.
31. V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703 of *LNCS*. Springer, 2007.
32. V. T. Vasconcelos, A. Ravara, and S. J. Gay. Session types for functional multithreading. In *CONCUR*, volume 3170 of *LNCS*. Springer, 2004.
33. J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE, 2000.
34. J. Villard, É. Lozes, and C. Calcagno. Proving copyless message passing. In *APLAS*, volume 5904 of *LNCS*. Springer, 2009.
35. P. Winterbottom. Alef language reference manual. In *Plan 9 Programmer's Manual: Volume Two*. AT&T Bell Laboratories, 1995.