# Proving correctness of Lace using interactive theorem prover tools

Thijs van Ede
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
t.s.vanede@student.utwente.nl

## 1. UPDATE

This week I started writing the introduction, which still needs a part on the background information. Perhaps, some explanation that is now at the Method section will be moved to the background information section. I wrote a new part on the method information about the order of linearisation points, and made some flow diagrams to illustrate this. The theorems to be proven are specified as invariants. Furthermore, I began documenting some proofs for Lace, which I will continue doing over the next week. When I've finished the documentation of the proof of Lace, I will start writing the conclusion and make some final additions to the paper. That is what I am planning to do this week. Furthermore, I made an appointment with my supervisor to show my progress and ask some questions.

## ABSTRACT

Interactive theorem prover tools

## 2. INTRODUCTION

Modern day processors' multi-core architecture has increased demand for concurrent programming to fully utilise the processing power. These concurrent programs need to be proven correct to ensure the desired execution. Several methods for proving programs correct exist such as separation logic[7] and first-order logic[9].

Interactive theorem prover tools such as Isabelle[6], Coq[1], and PVS[5] use various methods of logical based proving check models. These tools are helpful in proving properties of both sequential[2] and concurrent[3] [8] algorithms. The theorem prover tool PVS uses several decision procedures and a symbolic model checker, in combination with a random tester to help generating formal proofs.

This paper tries to proof several properties of the Lace algorithm[10] to illustrate the method of proving concurrent programs. Lace is a concurrent algorithm for thread scheduling among processors. Its use of a memory fence, compare-and-swap system call and both concurrent and sequential components make it an ideal algorithm to proof correct.

In this paper, we will prove the Lace algorithm correct using the interactive theorem prover tool PVS. The algorithm is modeled by constructing linearisation points where the state of the system changes, and using them to generate a flow diagram of the program. Subsequently, PVS assists in proving the required invariants using this model in combination with several assumptions about the system. With the invariants a comprehensive proof of the Lace algorithm is constructed.

### 2.1 Background information

## 3. PRELIMINARIES

### 3.1 System

This paper assumes the algorithm runs on a shared memory system with the x86 memory model. This memory model allows the reordering of loads before stores, that is, write operations are buffered before they are stored in memory. Thus threads might read old values, even though they have been written by other threads. These writes may not have become globally visible yet, because they are still buffered. Memory fences are used to flush these write buffers and make changes globally visible. Memory stores are immediately visible to threads that wrote them, hence they do not require memory fences.

### 3.2 Compare-and-swap

A compare-and-swap (`cas`) ensures thread safety by simulating an atomic memory operation. The `cas` operation takes three parameters as input, namely a `variable`, an `expected value`, and a `replacement value`. The `cas` operation compares the value of `variable` to the `expected value`. When those values are equal, the `replacement value` replaces the current value of `variable` and `cas` returns `true`. If they are not equal, `variable` retains its current value and `cas` returns `false`. Because of its atomicity, `cas` ensures only one thread executes successfully when multiple threads invoke `cas` with the same valid parameters. `cas` is used for concurrent programming as it ensures thread safety and provides feedback to executing threads on whether their operation succeeded or not.

### 3.3 Linearisation points

An important aspect of proving concurrent algorithms is defining linearisation points[4]. These linearisation points are the points in time the state of the system changes. When reasoning about concurrent programs, information about the state of the system is needed to predict its expected behaviour. By defining linearisation points, the order in which they occur can be varied and thereby all possible states can be derived. That is, by varying the order of linearisation points, all possible equivalent sequential programs can be derived, which in turn can be reasoned about.

## 3.4 Assumed properties

First of all, we have four ways of classifying a task.

Definition 1. *A task $x$ exists* $\iff x <$ `head`

Definition 2. *A task $x$ is stolen* $\iff x <$ `tail`

Definition 3. *A task $x$ is shared* $\iff x <$ `split`

Definition 4. *A task $x$ is private* $\iff x \geq$ `split`

In proving Lace correct, this paper assumes there is only one owner thread per deque and a possibly infinite number of thieving threads. In practice there are more than one owner threads since there are multiple deques, however, the proof of correctness for one deque is equivalent to all other deques. I.e. we assume that owner threads are not influenced by each other, only by thieving threads.

The second assumption is that thieving threads only call `steal` and the owner threads only calls `push` and `pop`. This implies that the owner thread is also the only thread that has access to `grow_shared` and `shrink_shared`, since they are only called in `push` and `pop`.

In addition, this paper assumes the number of calls to `pop` is equal to the number of calls to `push` when the algorithm finished executing. I.e. the deque is empty when the program finished.

Finally, all deque values - i.e. tail, split, head, and the entries within the deque - are all set to zero when the deque is initialised.

1. One owner thread per deque.
2. Possibly infinite number of thieving threads per deque.
3. `steal` is only called by thieving threads.
4. `push` and `pop` are only called by owner thread.
5. Owner thread only calls `grow_shared` and `shrink_shared`.
6. Number of `pop` calls $\leq$ number of `push` calls.
7. When finished, number of `pop` calls = number of `push` calls.
8. On initialisation all deque variables are set to zero.

**Figure 1: Assumed properties of Lace**

## 3.5 Required properties

Lace requires some properties to work correctly. First of all, the variables `head`, `split`, and `tail` should stay within bounds. Figure 2 expresses this requirement as invariants. Furthermore, a task must be stolen only once, and executed exactly once. To prove these properties, they are written as invariants. When every task is stolen only once, it means thieving threads can only steal tasks $\geq$ `tail` and that when tail reduces, a task cannot be stolen again. The former part of the property is trivial, if tail increases it steals an unstolen task which is consistent with the requirement. The latter part is less trivial, if tail reduces, a stolen task cannot be executed again, therefore the stolen tasks need to be removed before tail is reduced. This is written as an invariant in Figure 3. To elaborate on this invariant: either `tail` increases when a new task is stolen, or `tail` decreases but the task has to be removed from the deque so that it cannot be stolen again. When a task is executed once, this implies it is only stolen once and not executed by the owner thread, or it is executed by the owner thread and never stolen. This can be written as the invariant in Figure 4.

```
0 ≤ head ≤ size
0 ≤ split ≤ head
0 ≤ tail ≤ head
```

**Figure 2: Invariants: variables stay within bounds.**

```
tail.new > tail.old || (tail.new ≤ tail.old &&
tail.old == head && pop() == STOLEN, head)
```

**Figure 3: Invariant: every task is stolen only once.**

```
(pop() == STOLEN && head.new < head.old) ||
(pop() == WORK && head.new < head.old)
```

**Figure 4: Invariant: every task is executed only once.**

## 4. METHOD

In proving a concurrent program such as Lace correct, the linearisation points of the algorithm must be appointed. For each of these linearisation points, the preconditions can be defined, keeping in mind the variables that can change because of concurrency. By altering the order in which the linearisation points occur, each possible state of the system can be derived. From here conclusions can be drawn about the desired behaviour of the system and its possible behaviour.

The first step of appointing linearisation points is trivial and could be done automatically. However, because the algorithm is relatively simple and to illustrate how these points can be identified it is done manually. First all global variables of the algorithm are identified, then each point in the algorithm where a variable is changed is appointed as a linearisation point.

The second step of defining the preconditions of each linearisation point is less trivial and for illustrative purposes they are also deduced manually. In this case, the preconditions consist of the state of the system before the linearisation point. This can be derived by the variables that are adjusted before the linearisation can occur and that cannot change due to other linearisation points. Either because other linearisation points do not modify the variable or because the linearisation points that do modify it do not occur.

After the preconditions are established, the linearisation point will alter the state of the system, from this it is possible to calculate the postconditions. However, due to concurrency, linearisation points can occur in different possible sequences, which will result in different outcomes for the state of the system. Conclusions are drawn from these states, which will result in the invariants that are necessary to prove the system works correctly. That is if it is possible to find invariants that state `tail.old ≤ tail.new || (tail.old == head && pop() && tail.new ≤ head)` and `(tail.old ≤ tail.new && pop() == STOLEN) || pop() == WORK && tail < head`.

To find out how threads and their manipulation of deque variables influence the distribution of work in the deque, the Lace algorithm has to be studied carefully. This part will mainly be a literature research, the Lace algorithm is described in detail in [10]. When the algorithm is modeled in PVS, an even more clear view ill become apparent on the influence of variables on the deque.

In proving Lace correct, we propose to model the Lace algorithm in the interactive theorem prover tool PVS[5]. Here, the algorithm must be mapped to a series of mathematical lemma's, theorems, and functions. Furthermore, the desired outcome must be expressed in set of theorems. In the case of Lace, these theorems include a task being executed once, and the task always giving the same result, independent of the process that executes it. After the algorithm has been modeled, PVS assists in generating a proof for the desired outcome. This is done by applying formal logic to the given formulas. Interactive theorem prover tools indicate whether the given model is enough to prove the algorithm correct. If not, it provides missing theorems that needs to be proven by the user to prove the entire algorithm correct.

Various interactive theorem prover tools may be used for proving this algorithm correct. However, this paper suggests to use the PVS 6.0 Allegro Lisp Binary for a Linux Intell 64-bit machine. The model of the algorithm is constructed from its description in the paper [10].

A proof constructed with an interactive theorem prover such as PVS holds in all cases and could also be constructed without using PVS. The tool only helps to construct the proof and formalise the method of proving algorithms correct. PVS will therefore speed up the process of constructing a proof and increase confidence in the proof. It also increases the transferability of this research since the tool constructs equal proofs in an identical way.

## 5. MODELING LACE

A model for the Lace algorithm is constructed from its specifications as described in [10] and displayed in Figure 14. Note the that Figure 14 projects a slightly different form of the algorithm. However, this alternate form is used to construct a model for Lace. First of all, a model is created from the algorithm without the `allstolen` flag.

### 5.1 Linearisation Points

In Lace, the state of the system is dependent on the variables head, split, and tail. The linearisation points are defined on these variables and can be found in Table 1a, 1b, and 1d respectively. Note that the variables used at linearisation points to change the state of the system are read at a point before they are used. Thus the value might have changed when the program reaches the linearisation point, in which case it still uses the old value.

In addition to these three main variables, the algorithm also uses a `movesplit` and `allstolen` variable. These variables influence the behaviour of the system by indicating whether the split point must be moved or all public tasks are stolen respectively. Therefore, these variables are also included in the linearisation points, which can be found in Table 1e, 1f, and 1g.

#### *5.1.1 Head*

The head variable is modified in both the push and the pop methods. Table 1a shows an overview of the linearisation points and the places where the variables used in the linearisation points are initialised. The first linearisation point is at line 13 of Lace as shown in Figure 14, where head is incremented by 1. In this case the variable head is read in the same line of code but does not occur simultaneously with the write operation. The second and third linearisation points are at line 45 and 47 of Lace as shown in Figure 14 respectively. In both cases, head is decreased by 1 and the variable is read in the same line of code as it is written. As with the first point, the read and write operation do not occur simultaneously.

#### *5.1.2 Split*

In Lace, the variable split has two instances, namely split and o_split. The latter instance of split is private, and thus visible to the owner thread but not to thieving threads. To avoid confusion, split and o_split are displayed in Table 1b and Table 1c respectively. Just as explained in section 5.1.1 these tables indicate the methods, linearisation points, operations, initialisation points, and variables for the variable scrutinised, in this case the split and o_split variables. As opposed to the linearisation points of the variable head, split has linearisation points in which variables are used that are read in a different line of the algorithm. This might indicate a greater chance of the variables being altered before they are used.

#### *5.1.3 Tail*

The final variable of the deque that is important to include in the model is tail, its linearisation points can be found in Table 1d. This variable is modified in the steal and push functions at lines 5 and 15 respectively. In steal, tail is read at line 3 whilst incremented at line 5. Because of this gap tail can be modified in the meantime by other threads. The same goes for the modification of tail in the push method, since read and write do not occur simultaneously.

#### *5.1.4 Movesplit*

Movesplit is not part of the deque, but it indicates whether the owner thread should grow the shared part of the deque. This boolean variable is set to true at line 8 of the algorithm and is set to false at lines 16 and 25. These linearisation points can be found in Table 1e.

#### *5.1.5 Allstolen*

As with split, the variable `allstolen` has two instances, namely `allstolen` and `o_allstolen`. The global boolean variable `allstolen` is set to false at line 17 of the algorithm and to true at line 39. The private `o_allstolen` is set to false and true at lines 19 and 40 respectively. The linearisation points can be found in Table 1f and 1g.

### 5.2 Preconditions

For all initialisation points, some preconditions about the state of the system can be assumed. In this section, the preconditions for each initialisation point as described in section 3.3 are defined and substantiated.

#### *5.2.1 Head*

The first thing to note about `head` is that it is only modified from within the methods `push` and `pop`. It is also assumed that these methods are only called from within the owner thread, implying there is no interference on that variable from other threads. The first instance of `head` being modified is at line 13 of the algorithm in the method `push`. Before the modification, line 11 checks whether `head == size`, if this is the case the method returns `FULL` and thus will not reach line 13. Therefore, at the linearisation point of line 13 it is certain that `head` does not equal `size`.

At line 45 in the method `pop`, `head` is decremented. At line 43 the method checks whether `head == 0`, if so the method returns. In combination with the assumption that the number of pops never exceed the number of pushes, we derive that at this point, head must be larger than 0. Because the if statement in line 44 must be true to reach line 45, `o_allstolen` is true, or `o_split == head` and `shrink_shared()` returns true. The method `shrink_shared` only returns true after it set `allstolen` and `o_allstolen`.

In combination with the assumption that both of these variables can only be adjusted by the owner thread, these will not change. Therefore at the linearisation point of **head** at line 45, **o_allstolen** is set.

At line 47 in **pop**, **head** is also decremented. When this case appears, **o_allstolen** is false and either **o_split !=** **head** or **shrink_shared** returned false or both. From this we conclude that **o_allstolen** is not set. //TODO check whether all variables needed are covered.

### 5.2.2 *split*

As with **head**, **split** and **o_split** can also only be modified within the methods reachable by the owner, in this case **push**, **grow_ shared**, and **shrink_shared**. At the first linearisation point at line 15, **o_allstolen** is set and **head** $\geq 1$. The latter statement is true since the number of **pop** calls do not exceed the number of **push** calls, and **head** is incremented by 1 at line 13.

At the second linearisation point at line 23, **movesplit** is set. Since **grow_shared** is only called when movesplit is set and movesplit cannot be unset by any other thread than the owner,

At the third linearisation point at line 30, $\texttt{t} \neq \texttt{s}$, note that these are the local values initialised at line 27. Thus **tail** and **split** can be altered.

The linearisation point at line 36 also implies $\texttt{t} \neq \texttt{s}$, because of the if-statement at line 33. The memory fence at line 31 secures the knowledge that **tail** is not modified until the memory fence is .

### 5.2.3 *tail*

**tail** is modified by both thieving threads and the owner thread. Notably, the first linearisation point of **tail** is at line 5 of the algorithm, where tail is increased. This modification uses the compare-and-swap method, which only executes if the given values are equal. In this case the values **t** and **s** read at line 3 must be equal to the current values of **tail** and **split** respectively. If this occurs, it can have two causes: neither values have changed, or the values have changed but are restored to their original state between line 3 and 5 of the algorithm.

At the second linearisation point at line 15, the flag **o_allstolen** is set, since this is the only way to reach this part of the code and **o_allstolen** is only modified by the owner thread itself. Additionally, **head** is not equal to size, and incremented by 1 relative to line 13.

## 5.3 Order of linearisation points

By assuming only one owner thread exclusively calls methods **push** and **pop**, all variables manipulated only at those methods will occur in order. Subsequently, this is true for all methods **grow_shared** and **shrink_shared** which are invoked only through **push** and **pop**. The natural order of linearisation points called by **push** and **pop** can be derived by creating a flow diagram. In this section, the order of linearisation points for **push** and **pop** are derived respectively.

First of all, the method **push** has a flow diagram as depicted in Figure 5. Here the first linearisation point is at line 13, where head is increased. If **o_allstolen** is set, the program continues with linearisation point 15. Then, if **movesplit** is true, it is set to false, otherwise it continues flowing through all linearisation points down to linearisation point 19 where **o_allstolen** is set to false. Alternatively, when **o_allstolen** is false at line 14 and **movesplit** is true at line 20, it invokes the method **grow_shared** which is depicted as the right part of the flow diagram.

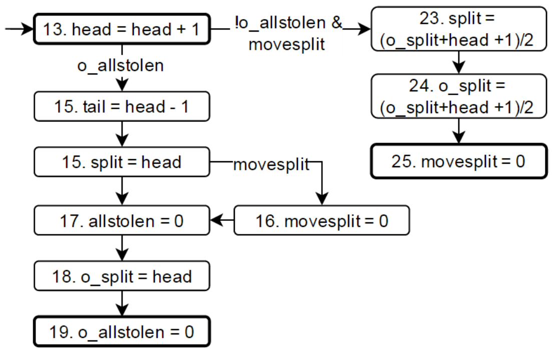

**Figure 5: Flow diagram of method push.**

Second, the method **pop** has a flow diagram as depicted in Figure 6. This diagram is less trivial than the **push** flow diagram. Since the method **pop** first checks whether head equals 0 and returns when it does the linearisation points are not reached, thus the first condition of $\texttt{head} \neq 0$ is introduced. After this, it checks whether **o_allstolen** is set, if this is the case, it continues to decrease head and returns. Alternatively, if **o_allstolen** is false, it checks whether **o_split == head**, if so, it calls the method **shrink_shared** following the rhombus' upper arrow. The method **shrink_shared** first checks whether the variables **t** and **s** that it read from **tail** and **split** respectively are equal. If this condition is true, it sets **allstolen** and **o_allstolen** after which it returns true and continues at **pop**. Subsequently, if the condition is false, it modifies **split** and sets a memory fence, depicted with the dashed lines. After reading the **tail** variable into **t** again, the algorithm checks whether **t** equals **s**. When they are equal, the method sets the **allstolen** variables, and returns true, after which it continues at **pop**. However, when they are not equal, **t** and the temporary variable **new_s** are compared. If $\texttt{t} > \texttt{new\_s}$, **split** is modified, otherwise only **o_split** is set to **new_s**. Thereafter, the method returns false and continues at **pop**.

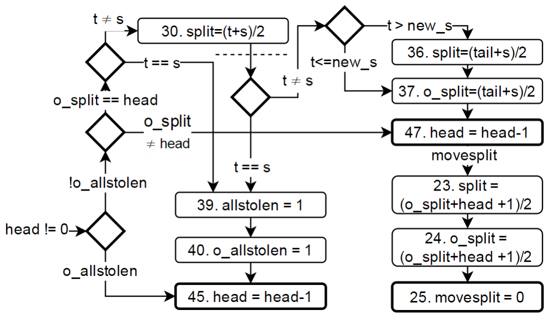

**Figure 6: Flow diagram of method pop.**

From these flow diagrams, the possible sequences of linearisation points are clear. Note that the linearisation points modified by thieving threads through the method **steal** are not included. Since these occur at arbitrary points in the algorithm, it does not make sense to include them in a flow chart.

## 6. PROVING LACE

LEMMA 1. $\textit{head} \leq \textit{size}$.

PROOF. Table 1a indicates head is only increased at method push. This indicates there is no need to take the reordering of loads before stores into account, since only one thread operates on this variable. Since it is assumed head $= 0$ at the start of the algorithm, i.e. head $<$ size, it is left to prove the linearisation point cannot be reached if head $\geq$ size. In PVS we input the code as depicted in Figure 7. To proof lemma lemma_head, we use axioms head_start, stating head $<$ size at the initialisation, and head_size stating head $\neq$ size, i.e. line 11 of Lace. Then, we expand function incr_head(head), which increases head by 1, i.e. line 13 of Lace. Then using PVS' grind option, it concludes that indeed the lemma holds. □

```
head_start:  AXIOM head < size
head_size:  AXIOM NOT (head = size)
incr_head(head):  posnat = head+1
lemma_head:  LEMMA incr_head(head) <= size
```

**Figure 7: PVS proof of head $\leq$ size**

LEMMA 2. *head* $\geq$ *0*

PROOF. Table 1a indicates head is only decreased at the method pop. As with Lemma 1, the reordering of loads before stores does not need to be taken into account, since the owner thread is the only thread operating on this variable. From the flow diagram we find that to reach a linearisation point where head is decreased, head $\neq$ 0. In PVS we input the code as depicted in Figure 8. Here we use the axiom head_start to denote that the starting position of head is 0, or in this case $\geq$ 0. The axiom head_zero states, head $\neq$ 0, as line 43 of Lace indicates. To proof lemma_head, we introduce both axioms and expand the method decr_head after which we use PVS' grind option. This concludes that indeed the lemma holds. □

```
head_start:  AXIOM head >= 0
head_zero:  AXIOM NOT (head = 0)
decr_head(head):  posnat = head-1
lemma_head:  LEMMA func_decr(head) >= 0
```

**Figure 8: PVS proof of head $\geq$ 0**

LEMMA 3. $\neg$ *allstolen* $\Rightarrow$ *tail* $\leq$ *split* $\leq$ *head*

PROOF. This paper assumes that the deque is initialised with tail = split = head = 0 and $\neg$ allstolen, hence, the lemma holds at at initialisation. Table 1f indicates allstolen is only reset at linearisation point 17. Figure 5 shows linearisation points 15 of tail and split must precede the reset of allstolen, which set tail and split to head-1 and head respectively. At this point the lemma still holds, and modifications of head, tail, and split need be scrutinised. First of all we note that tail might be modified in between linearisation points 15 and linearisation point 17.

Table 1d demonstrates variable tail is modified at linearisation point 5 and 15. Lemma 9 states that for linearisation point 5 to execute t < s and t == tail and s = split. When modeling this in PVS as in Figure 9, we see that tail $\leq$ split holds. Linearisation point 15 holds for this lemma since both tail and split are modified simultaneously to head-1 and head respectively.

Table 1b □

```
incr_tail(tail):  posnat = tail+1
lem_tail:  LEMMA tail < split IMPLIES
incr_tail(tail) <= split
```

**Figure 9: PVS proof of Lemma 3**

LEMMA 4. *allstolen = o_allstolen at load operations.*

PROOF. Table 1f and 1g indicate that both allstolen variables only change through methods push and shrink_shared. Figure 5 states that private variable o_split is the only variable to change in between the modifications to false of the allstolen variables. Subsequently, Figure 6 states there are no linearisation between the modifications to true at all. There do not occur any load operations of allstolen or o_allstolen in between either modification of the variables. The assumed memory model where loads can be reordered before stores does not influence this Lemma, since Table 1f states stealing threads only use allstolen. The owner thread does not reorder both allstolen variables, because the thread itself is the only thread modifying them. □

LEMMA 5. *split = o_split at load operations.*

PROOF. Table 1b and 1c affirm that both split variables only change through methods push, grow_shared, and shrink_shared. Figure 5 indicates that private variable movesplit and allstolen are modified between the linearisation points of the split variables. Furthermore, Figure 6 states that between linearisation points 23 and 24 no reads of either split variable occur. Between linearisation point 30 for split and 37 for o_split there are no read operations for either split variable. In addition, between linearisation point 36 of split and linearisation point 37 of o_split there are no read operations on split. The x86 memory model where loads can be reordered before stores is not important for this Lemma, since the method steal only uses split. Whereas the owner thread uses o_split, but loads cannot be reordered before stores for the thread that executes the stores. □

LEMMA 6. *shrink_shared returns True* $\Rightarrow$ *allstolen and o_allstolen.*

PROOF. Table 1f and 1g indicate both allstolen variables can only be modified within the methods push and shrink_shared. Since both can only be called by the owner thread, they cannot be run simultaneously. Thus, only shrink_shared needs to be inspected. The return true statement of shrink_shared corresponds to linearisation point 40 in Figure 6, since no more linearisation points occur between that point and the return true statement. Therefore, the code depicted in Figure 10 is used to proof the allstolen variables are set when shrink_shared returns true. By adding lemmas allstolen_set and o_allstolen_set to the proof, and using PVS' grind option, the tool verifies both allstolen flags are set to true if shrink_shared returns true. Note that stealing threads might not receive this modification of allstolen before they read the value since this paper assumes a system with the x86 memory model. □

LEMMA 7. *tail.new* $\geq$ *tail.old or (tail.new < tail.old and tail.old-tail.new* $\leq$ *#pop())*

PROOF. The first part of the lemma states that tail should increase, this is true for all linearisation points except at line 15 of Lace, as described in Table 1d. The second part states, that if tail is decreased, the method pop

```
allstolen_set:  AXIOM allstolen
o_allstolen_set:  AXIOM o_allstolen
shr_shared_ret:  LEMMA allstolen AND o_allstolen
```

**Figure 10: PVS proof of shrink_shared ⇒ allstolen && o_allstolen**

must have returned `STOLEN` and on modification, `tail.old` must have been equal to `head`. The only possibility for `tail` to decrease is at line 15 of Lace, where it is set to `head-1`, see Table 1d. As Figure 5 shows the linearisation point of line 15 can only be reached if `o_allstolen` is set.

□

LEMMA 8. *`pop()` ⟺ `head.new = head.old - 1`*

PROOF. Table 1a shows `head = head-1` can only be invoked from within the method `pop`, hence `head.new = head.old-1` ⇒ `pop()` holds. Figure 6 shows it is impossible to go from linearisation point 45 to 47 or vise versa. Furthermore, the Figure shows both linearisation points are the first possible end states to reach. However, the Figure does state that neither linearisation points are reached when `head == 0`. Though this is not aplicable to this situation since we assume the number of `pop` calls never exceed the number of `push` calls. Therefore `pop()` ⇒ `head.new = head.old -1` holds. □

LEMMA 9. *`tail.new = tail.old+1` ⇒ `!allstolen` and `t < s` and `t == tail` and `s == split`*

PROOF. The only linearisation point where `tail` is increased is at the `cas` operation of line 5 as stated in Table 1d. This modification only succeeds if `cas` succeeds, i.e. if the compare-and-swap method returns true. Thus this can be modeled in PVS as described in Figure 11. □

```
cas(var1, var2, exp1, exp2, val1, val2:  posnat)
:  bool = IF (var1 = exp1 AND var2 = exp2 THEN
true ELSE false ENDIF

steal(t, s, tail, split, allstolen):  bool =
IF allstolen THEN false ELSE (IF t < s THEN
cas(t,s,tail,split,(t+1),s) ELSE false ENDIF)
ENDIF

lem_incr_tail:  LEMMA (NOT allstolen AND
t < s AND t = tail AND s = split) IMPLIES
steal(t,s,tail,split,allstolen) = true
lem_incr_tail:  LEMMA NOT (NOT allstolen AND
t < s AND t = tail AND s = split) IMPLIES
steal(t,s,tail,split,allstolen) = false
```

**Figure 11: PVS proof of incrementing tail**

LEMMA 10. *`head.new = head.old - 1` and `pop() = STOLEN` ⇒ `allstolen`*

PROOF. Figure 6 illustrates `pop = STOLEN` in combination with the decrement of `head` as linearisation point 45, which can only be reached if `o_allstolen` or (`o_split == head` and `shrink_shared = true`. `o_allstolen`). Therefore `pop = STOLEN` ⇒ `o_allstolen`. This can be coded into PVS as illustrated in Figure 12. Here it is assumed `head` is decremented and `STOLEN` is returned if the if-statement of Lace rule 44 is true. Using PVS' grind option we find this lemma holds. From Lemma 6 we find `o_allstolen` or (`o_split == head and allstolen` and In combination with Lemma 4 stating that the `allstolen` variables are equal when at their load operations we conclude that `pop = STOLEN` ⇒ `allstolen`. □

```
shrink_shared(o_allstolen, allstolen):  bool =
o_allstolen AND allstolen

pop(head, o_split, o_allstolen, allstolen):
bool = NOT (head = 0) AND (o_allstolen OR
shrink_shared(o_allstolen, allstolen))

lem_allstolen:  LEMMA pop(head, o_split,
o_allstolen, allstolen) IMPLIES o_allstolen
```

**Figure 12: PVS proof pop() = STOLEN implies allstolen**

LEMMA 11. *`shrink_shared` is invoked ⟺ ¬ `allstolen`.*

PROOF. Figure 6 states `shrink_shared`, represented as linearisation points 30, 36, and 37, can only be reached if ¬`allstolen`. □

LEMMA 12. *`allstolen` ⇒ `tail` ≥ `split`*

PROOF. Table 1f indicates `allstolen` is only set to `true` in the method `shrink_shared`. Lemma 11 states that the variable is `false` when the method is invoked. `shrink_shared` sets `allstolen` to `true` if either at line 28 of Lace `t == s` or at line 33 `t == s`. □

LEMMA 13. *Linearisation point 23 ⇒ `split.new` ≥ `split.old`*

PROOF. □

LEMMA 14. *Linearisation point 30 ⇒ `split.new` ≤ `split.old`*

PROOF. Lemma `lem_decr_a` in Figure 13 corresponds to line 30 of Lace and states that `decr_tail` ≤ `split`, under the assumptions that `tail != split` and `tail < split`. The former assumption is valid since Figure 6 states the variables `tail` and `split` are not equal when linearisation point 30 occurs and the read variables `tail` and `split` are not modified in between. The latter assumption is valid since Lemma 9 states `tail` cannot increase if `tail` ≥ `split` and at the start of Lace `tail == split == 0`. Using PVS' grind option, this Lemma is proven correct. □

LEMMA 15. *Linearisation point 36 ⇒ `split.new` ≤ `split.old` where `split.old` is before Linearisation point 30.*

PROOF. Lemma `lem_decr_b` in Figure 13 corresponds to the linearisation point of line 36 of Lace and states that after the linearsation point of line 30, line 36 can increase `split`, but not beyond the point of `split.old`, i.e. `split` before the linearisation point of line 30. It assumes `tail != split` and `t > decr_tail(tail,split)` and `t < split`. The first of these assumptions is true according to Figure 6, it shows `t != s` is a requirement for reaching linearisation point 36. Since `tail` is reread at line 32, but `split` is equal to the value read at line 27 this assumption still holds. The second assumption is valid since Figure 6 expounds `t > new_s` for linearisation point 36 to be reached. That is, the newly read value of `tail` here modeled as `t` is larger than the computed value of `split` at linearisation point 30. Note that the newly read value of `tail` cannot increase any further, since the memory fence of line 31 asserts the modification of `split` is globally visible. Any reordering of read and write operations as the x86 memory model allow do not apply in this part of the code since the memory fence forces global visibility. The final assumption of `t < split` is true since Lemma 9 affirms `tail` cannot increase unless it is smaller than `split`. With these assumptions, PVS shows that indeed the newly computed value of `split.new` $\leq$ `split.old`. ☐

LEMMA 16. *`min(linearisation point 30, linearisation point 36)` is globally visible.*

PROOF. Lemma `lem_decr_c` in Figure 13 insists linearisation point 36 cannot decrease with respect to linearisation point 30. Here the assumptions of `lem_decr_a` and `lem_decr_b` are both needed to prove the lemma correct. PVS proves that the minimum value of split after linearisation point 30 and linearisation point 36 is linearisation point 30. This change is visible because of the memory fence of line 31. No memory fence is needed for linearisation point 36, since stealing threads cannot abuse this modification because `split.old` $\leq$ `tail.new`. Therefore stealing threads cannot steal more because of the modification. ☐

```
decr_tail(tail, split):  posnat = (tail+split)/2

lem_decr_a:  LEMMA NOT tail = split AND tail <
split IMPLIES decr_tail(tail,split) <= split

lem_decr_b:  LEMMA NOT tail = split AND t >
decr_tail(tail,split) AND t < split IMPLIES
decr_tail(t, decr_tail(tail,split)) <= split

lem_decr_c:  LEMMA NOT tail = split AND tail
< split AND t > decr_tail(tail, split) AND
t < split IMPLIES decr_tail(tail, split) <=
decr_tail(t, decr_tail(tail,split))
```

**Figure 13: PVS proof of Lemmas 14, 15, and 16. Method decr_tail returns the modified value of tail. lem_decr_a is used in Lemma 14, lem_decr_b is used in Lemma 15, and lem_decr_c is used in Lemma 16.**

LEMMA 17. *`allstolen` ⇒ ¬ `tail.new = tail.old + 1`*

PROOF. Table1d affirms `tail` can only increase at linearisation point 5 of Lace and Lemma 9 states that `allstolen` must be false to increase `tail`. Table 1f shows that `allstolen` can only be set to true at line 39 of Lace. However, the x86 memory model might not have globalised the modification of `allstolen`, or has globalised the variable after the stealing method checked it. Subsequently, the stealing thread reads `tail` and `split` into `t` and `s` respectively, and checks whether `t < s`, a requirement which is also stated in Lemma 9.

Figure 6 expounds `allstolen` can be modified if either of two preceding paths are taken. First, if `t == s`, this means

From Figure 6 we deduce that `allstolen` ⇒ `t == s`. Furthermore, Figure 6 `states that after the that is tail == split` ☐

```
 1 def steal():
 2   if allstolen: return NOWORK
 3   (t,s) = (tail,split)  #t, s are local
 4   if t < s:
 5     if cas((tail,split), (t,s), (t+1,s)):
 6       return WORK(t)
 7     else: return NONE #busy
 8   elif not movesplit: movesplit = 1
 9   return NONE #no work

10 def push(data):
11   if head == size: return FULL
12   #write task data at deque head
13   head = head + 1
14   if o_allstolen:
15     (tail,split) = (head-1,head)
16     if movesplit: movesplit = 0
17     allstolen = 0
18     o_split = head
19     o_allstolen = 0
20   elif movesplit: grow_shared()

21 def grow_shared():
22   new_s = (o_split+head+1)/2
23   split = new_s
24   o_split = new_s
25   movesplit = 0

26 def shrink_shared():
27   (t,s) = (tail,split)
28   if t != s:
29     new_s = (t+s)/2
30     split = new_s
31     MFENCE
32     t = tail #read again
33     if t != s:
34       if t > new_s:
35         new_s = (t+s)/2
36         split = new_s
37       o_split = new_s
38       return FALSE
39   allstolen = 1
40   o_allstolen = 1
41   return TRUE

42 def pop():
43   if head == 0: return EMPTY, None
44   if o_allstolen or (o_split == head and
         shrink_shared()):
45     head = head-1
46     return STOLEN, head
47   head = head-1
48   if movesplit: grow_shared()
49   return WORK, head
```

**Figure 14: Lace algorithm as described in "Lace: Non-blocking Split Deque for Work-Stealing"[10]**

## 7. REFERENCES

[1] The coq proof assistant. https://coq.inria.fr/what-is-coq. Accessed: 2015-03-29.

[2] B. Badban, W. Fokkink, J.F. Groote, J. Pang, and J. van de Pol. Verification of a sliding window protocol in $\mu$crl and pvs. *Formal Aspects of Computing*, 17(3):342–388, 2005.

[3] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Computer Aided Verification*, pages 475–488. Springer, 2006.

[4] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[5] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *Automated Deduction - CADE-11*, pages 748–752. Springer, 1992.

[6] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[7] 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.

[8] N. Shankar. Verification of real-time systems using pvs. In *Computer Aided Verification*, pages 280–291. Springer, 1993.

[9] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[10] T. van Dijk and J.C. van de Pol. Lace: Non-blocking split deque for work-stealing. *Lecture Notes in Computer Science*, 8806:206–217, 2014.

| method | lin pt | op | init pt | var |
|---|---|---|---|---|
| push | 13 | head+1 | 13 | head |
| pop | 45 | head-1 | 45 | head |
| pop | 47 | head-1 | 47 | head |

**(a) Linearisation points of head variable.**

| method | lin pt | op | init pt | variables |
|---|---|---|---|---|
| push | 15 | head | 15 | head |
| gr_shared | 23 | new_s=(o+h+1)/2 | 22 | o_split,head |
| shr_shared | 30 | new_s=(t+s)/2 | 27 | tail,split |
| shr_shared | 36 | new_s=(t+s)/2 | t:32,s:27 | tail,split |

**(b) Linearisation points of split variable.**

| method | lin pt | op | init pt | variables |
|---|---|---|---|---|
| push | 18 | head | 18 | head |
| gr_shared | 24 | new_s=(o+h+1)/2 | 22 | o_split,head |
| shr_shared | 37 | new_s=(t+s)/2 | 27 | tail,split |
| shr_shared | 37 | new_s=(t+s)/2 | t:32,s:27 | tail,split |

**(c) Linearisation points of o_split variable.**

| method | lin point | op | init point | variables |
|---|---|---|---|---|
| steal | 5 | t+1 | 3 | tail |
| push | 15 | head-1 | 15 | head |

**(d) Linearisation points of tail variable**

| method | lin point | op | init point | variables |
|---|---|---|---|---|
| steal | 8 | true | 8 | movesplit |
| push | 16 | false | 16 | movesplit |
| grow_shared | 25 | false | 25 | movesplit |

**(e) Linearisation points of movesplit variable**

| method | lin point | op | init point | variables |
|---|---|---|---|---|
| push | 17 | false | 17 | allstolen |
| shrink_shared | 39 | true | 39 | allstolen |

**(f) Linearisation points of allstolen variable**

| method | lin point | op | init point | variables |
|---|---|---|---|---|
| push | 19 | false | 19 | o_allstolen |
| shrink_shared | 40 | true | 40 | o_allstolen |

**(g) Linearisation points of o_allstolen variable**

**Table 1: The method column indicates the method where the variable is modified. Lin pt column refers to the linearisation point in the Lace algorithm as shown in Figure 14. The op column explains the executed operation. The init pt column refers to the point in the Lace algorithm where the variables are read. The final column var declares the variables that are used to modify variable. Note that in 1b and 1c o_split and head are abbreviated to o, h respectively. Tail and split are abbreviated to t and s since these is a local variables as in the algorithm.**

| lin point | head | allstolen |
|---|---|---|
| 13 | $\neq$ size | |
| 45 | >0 | true |
| 47 | >0 | true |

**(a) Preconditions head variable**

| lin point | head | tail | split | allstolen |
|---|---|---|---|---|
| 5 | | t=tail | s=split | |
| 15 | $\neq$size | >0 | true | true |

**(b) Preconditions points tail variable**

**Table 2: Full caption**