Model checking simulation rules for linearizability

Graeme Smith

School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

Abstract. Linearizability is the standard notion of correctness for concurrent objects. A number of approaches have been developed for proving linearizability along with associated tool support. In this paper, we extend the tool support for an existing simulation-based method. We complement the current theorem-prover support with model checking to allow a means of quickly finding problems with an implementation before attempting a full verification. Our model checking approach is novel in that it is used to verify the simulation rules, rather than directly trying to check an object being accessed by a number of threads. As a consequence, verification can be done for an arbitrary number of accessing threads; something that is not possible with existing approaches based on model checking.

1 Introduction

Concurrent objects are objects which have been designed to allow simultaneous access by more than one thread. They include locks and data structures, and are common in modern software libraries such as `java.util.concurrent`. They may employ *coarse-grained locking*, where one thread locks the object forcing all others to wait, but for efficiency are more likely to employ *fine-grained locking*, where only parts of the object are locked, e.g., two adjacent nodes in a linked list, or *non-blocking algorithms*, where no locking is employed [11]. In the cases of fine-grained locking and non-blocking algorithms, lines of the object's code being executed by different threads are interleaved leading to subtle behaviour that is difficult to verify.

The main notion of correctness for concurrent objects is *linearizability* [12]. It compares an abstract specification of a concurrent object, where all operations are atomic, and a concrete specification (or implementation), where operations may overlap. It requires that each operation of the concrete specification *appears* to take place atomically at some point between its invocation and return – the operation's *linearization point* [12] – and that the resulting sequence of such points corresponds to a sequence of operations on the abstract specification. Effectively this means that overlapping concrete operations can occur in any order in the abstract sequence, but when one concrete operation returns before another is invoked that order must be preserved in the abstract sequence.

A number of approaches have been developed for proving linearizability along with associated tool support [1, 4, 24, 9, 7, 8, 18]. In particular, Derrick et al. [7, 8, 18] have developed a simulation-based method for proving linearizability supported by the interactive theorem prover KIV [17]. This approach has been proved sound and complete, the soundness and completeness proofs themselves being done in KIV.

Although not automatic, a strength of Derrick et al.'s approach is the fact that, being based on theorem proving, the size of the concurrent object's state space is not restricted, and verification can be done for an arbitrary number of accessing threads. This is not possible with existing approaches based on model checking where both the size of the data structure, and the number of threads needs to be restricted [26, 25, 2, 14, 28, 22]. In each model checking approach, the size of stacks and queues is limited to between 2 and 5 items. In all approaches other than [28], the number of threads is limited to between 2 and 4 (depending on the complexity of the object). In [28], which uses partial-order reduction and symmetry reduction to increase the number of threads, that number is still limited to between 3 and 6 for the objects verified.

In this paper, we provide a model checking approach that, while similarly limited in terms of state space, allows an arbitrary number of threads. This is achieved by using the model checker to verify the simulation rules of Derrick et al.'s approach, rather than trying to directly check an object being accessed by a number of threads. The approach is intended to complement, rather than act as a replacement for, the use of KIV. In particular, it is intended to be used as a means of quickly finding problems with implementations before attempting a full verification in KIV.

We show how the approach is encoded in TLC [27], the model checker for TLA+ [13],[1] but other state-based model checkers, e.g. SAL [5], could be used. We do not try to optimise the model checking; this paper is a proof of concept and we leave the development of an efficient tool to future work.

The paper is structured as follows. In Section 2, we introduce the simulation rules of Derrick et al. and our running example, the Treiber stack [23]. In Section 3, we show how the simulation rules can be encoded in TLC when the abstract specification's operations are deterministic; as argued by Burkhardt et al. [2] this is nearly always the case. For completeness, we provide an alternative encoding to handle cases where the abstract specification has one or more non-deterministic operations in Section 4. We conclude with a discussion of future work in Section 5.

2 Simulation-based proof method

The work of Derrick et al. [7, 8, 18] identifies different proof rules for use with 3 classes of linearizability proofs of increasing complexity. The first and simplest class of proofs are those where each operation's linearization point can be determined from the current state of the calling thread and object [7]. The next class

[1] This choice was partly inspired by the use of TLA+ and TLC at Amazon [15, 16].

involves operations whose linearization points are determined by future states, possibly resulting from the operations of other threads [8]. The final class includes objects whose linearization points can only be determined by examining the whole global history [18].

In the first two cases, the proof rules reduce reasoning about an arbitrary number of processes to thread-local reasoning about one process and its environment which is abstracted to one other process. In the latter case, proving linearizability is reduced to finding a backward simulation relation between simple extensions of the abstract and concrete specifications of the concurrent object. This latter approach is complete in itself, but is generally more difficult to apply than the approaches for the first two cases. In all cases, proofs are *step-local* meaning reasoning is performed on one line of code at a time.

In this paper, we focus on the first class of proofs. Extending our work to the other classes is discussed in Section 5.

2.1 The Treiber stack

To illustrate the proof method and our model checking approach in the rest of the paper, we introduce as a case study the Treiber stack [23]. The Treiber stack was the first proposed non-blocking implementation of a concurrent list-based stack. A typical implementation (taken from [7]) is given below, where `Node` is a class with two fields `val:T` and `next:Node`, and `T_empty` is the type `T` augmented with the additional value `empty`.

```
head : Node;   \\ global variable
n, ss, ssn : Node;  lv:T;   \\ thread-local variables
```

```
push(v : T) :
n = new(Node);
n.val = v;
repeat
    ss = head;
    n.next = ss;
until CAS(head, ss, n)
return;
```

```
pop() : T_empty
repeat
    ss = head;
    if ss = null
        return empty;
    ssn = ss.next;
    lv = ss.val;
until CAS(head, ss, ssn);
return lv;
```

A thread doing a `push` operation assigns the value being pushed onto the stack to the `val` variable of a new node stored in local variable `n`. It then repeatedly tries to make `n` the head of the stack by setting a local variable `ss` to the global variable `head`, setting `n`'s `next` variable to `ss`, and then assigning `head` to `n` provided it is still equal to `ss` (i.e., provided another thread has not in the meantime changed the value of `head`). `CAS(a,b,c)` is an atomic operation (supported by most microprocessors) which compares `a` and `b` and, if they are equal, sets `a` to `c` and returns true; otherwise it leaves `a` unchanged and returns false.

A thread doing a `pop` operation repeatedly sets `ss` to `head`, returning `empty` if `ss` is null, and otherwise setting `ssn` to `ss`'s `next` variable and local variable `lv` to `ss`'s `val` variable and, finally, assigning `ssn` to `head` and returning `lv` provided `head` is still equal to `ss`.

The Treiber stack is linearizable with respect to the following abstract specification of a stack (given in Z [21][2]). The linearization point of `push` is the final `CAS` which returns true. The linearization point of `pop` is either line 7 (when `ss` is assigned `null`), or the final `CAS` which returns true.

$[T]$

$$\begin{array}{|l}\hline \textit{AS} \\ \hline \textit{stack} : \operatorname{seq} T \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{ASInit} \\ \hline \textit{AS} \\ \hline \textit{stack} = \langle\,\rangle \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{Push} \\ \hline \Delta \textit{AS} \\ v? : T \\ \hline \textit{stack}' = \langle v? \rangle \frown \textit{stack} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{Pop} \\ \hline \Delta \textit{AS} \\ v! : T \cup \{\textit{empty}\} \\ \hline \textit{stack} = \langle\,\rangle \Rightarrow \\ \quad v! = \textit{empty} \wedge \textit{stack}' = \textit{stack} \\ \textit{stack} \neq \langle\,\rangle \Rightarrow \textit{stack} = \langle v! \rangle \frown \textit{stack}' \\ \hline\end{array}$$

We use set union to add the special value *empty* to the type T in operation *Pop* although strictly this should be done using a free type definition in Z [21].

2.2 Simulation-based proof

To apply the approach of [7], we first need to derive a concrete Z specification from the implementation. This specification has one or two operations for each line of code. The state is described by two schemas representing the global and thread-local variables. For the Treiber stack, the global state GS includes a variable *head* and the shared memory in which nodes are stored. Let *Ref* be the set of all references to nodes, and T be the type of values in a node.

$$\begin{array}{|l}\hline \textit{GS} \\ \hline \textit{head} : \textit{Ref} \cup \{\textit{null}\} \\ \textit{mem} : \textit{Ref} \nrightarrow (T \times (\textit{Ref} \cup \{\textit{null}\})) \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{GSInit} \\ \hline \textit{GS} \\ \hline \textit{head} = \textit{null} \\ \textit{mem} = \varnothing \\ \hline\end{array}$$

The local state LS includes the variables n, ss, ssn, lv and v (the input variable) appearing in the code, as well as a variable pc denoting the program counter. Let $PC == 0..13$ where 0 denotes that the thread is idle, i.e., not executing an operation.

[2] Following [7] we adopt the blocking semantics of Z in which operations are *guarded*, i.e., unable to occur when their predicate cannot be satisfied [6].

$$
\begin{array}{|l}
\multicolumn{1}{l}{LS} \\
\hline
n, ss, ssn : Ref \\
lv, v : T \\
pc : PC \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{LSInit} \\
\hline
LS \\
\hline
pc = 0 \\
\hline
\end{array}
$$

For each operation, there is an invocation operation which requires pc to be 0 and sets it to the first line of the operation.[3]

$$
\begin{array}{|l}
\multicolumn{1}{l}{Push0} \\
\hline
\Xi GS \\
\Delta LS \\
v? : T \\
\hline
pc = 0 \land v' = v? \land pc' = 1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Pop0} \\
\hline
\Xi GS \\
\Delta LS \\
\hline
pc = 0 \land pc' = 7 \\
\hline
\end{array}
$$

Then for each non-branching line of code there is a single operation. For example, for lines 2 and 3 we have

$$
\begin{array}{|l}
\multicolumn{1}{l}{Push2} \\
\hline
\Delta GS \\
\Delta LS \\
\hline
pc = 2 \land pc' = 3 \\
mem'(n) = (v, second(mem(n))) \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Push3} \\
\hline
\Xi GS \\
\Delta LS \\
\hline
pc = 3 \land pc' = 4 \\
ss' = head \\
\hline
\end{array}
$$

For each branching line of code there are 2 operations. For example, for line 5 we have

$$
\begin{array}{|l}
\multicolumn{1}{l}{Push5t} \\
\hline
\Delta GS \\
\Delta LS \\
\hline
pc = 5 \land head = ss \land pc' = 6 \\
head' = n \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Push5f} \\
\hline
\Xi GS \\
\Delta LS \\
\hline
pc = 5 \land head \neq ss \land pc' = 3 \\
\hline
\end{array}
$$

Following the approach of [7], we then have two proof obligations for each operation of the concrete specification.

Step 1. Firstly, we need to show that the lines of code defining the concrete operations simulate the abstract operations. To do this, we identify one line of code as the *linearization step*. This line of code must simulate the abstract operation, all others simulating an abstract skip. For example, for the operation `push` we require that line 5 simulates the abstract operation when `head` equals `ss`, and all other lines simulate an abstract skip (see Figure 1 for a possible execution of the operation).

To do this we need to define an abstraction relation relating the global concrete state space gs and abstract state space as. The abstraction relation ABS for the Treiber stack is defined recursively as follows.

[3] Following [7], we assume all values of variables and values in the range of functions that are not explicitly changed by a Z operation, remain unchanged.

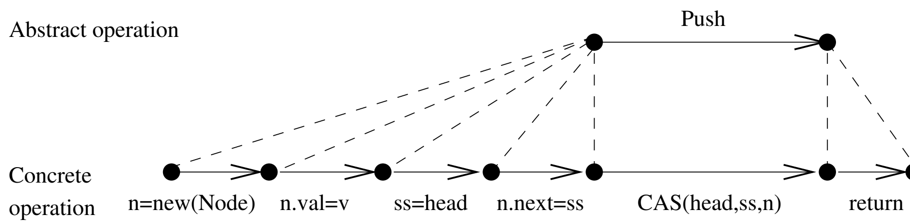

Fig. 1. Simulation of *Push*

$$
\begin{aligned}
&ABS(as, gs) == ABS0(as.stack, gs.head, gs.mem)\\
&ABS0(s, h, m) == (h = null \Rightarrow s = \langle\,\rangle) \wedge\\
&\qquad\qquad\qquad (h \neq null \Rightarrow s \neq \langle\,\rangle \wedge h \in \operatorname{dom} m \wedge first(m(h)) = head(s)\\
&\qquad\qquad\qquad\qquad \wedge ABS0(tail(s), second(m(h)), m)
\end{aligned}
$$

We also need to define an invariant to enable the simulation of each line of code to be proven independently. In our example, to prove that the line of code `CAS(head,ss,ssn)` simulates the effect of the abstract operation when `head` equals `ss`, this invariant needs to ensure that when pc=5 and $head = ss$ we have

$$
\begin{aligned}
&n \in \operatorname{dom} mem \wedge first(mem(n)) = v \wedge second(mem(n)) = ss \wedge\\
&(\forall r : \operatorname{dom} mem \bullet second(mem(r)) \neq n) \wedge ss \neq n
\end{aligned}
$$

The second line of this predicate ensures that n is a new node not referenced by any other.

Such an invariant is stated in terms of the global concrete state space gs and the local concrete state space ls. Hence, the invariant $INV(gs, ls)$ must imply the following.

$$
\begin{aligned}
&ls.pc = 5 \wedge gs.head = ls.ss \Rightarrow ls.n \in \operatorname{dom} gs.mem \wedge\\
&\quad first(gs.mem(n)) = ls.v \wedge second(gs.mem(n)) = ls.ss) \wedge\\
&\quad (\forall r : \operatorname{dom} gs.mem \bullet second(gs.mem(r)) \neq ls.n) \wedge ls.ss \neq ls.n
\end{aligned}
$$

Each simulation is then proved by one of 5 rules depending on whether the line of code is an invocation (beginning an operation), return (ending an operation) or internal step (neither an invocation nor return), and whether it occurs before or after the linearization step. A function $status(gs, ls)$ is defined to identify the linearization step. Before invocation, $status(gs, ls)$ is $IDLE$. After invocation but before the linearization step it is equal to $IN(in)$, where $in : In$ is the input to the abstract operation, and after the linearization step it is equal to $OUT(out)$, where $out : Out$ is the output of the abstract operation. The types In and Out have a special value $\perp$ denoting no input or output, respectively. As well as identifying the linearization point, the *status* function is used to store the input of the invocation step until it is needed at the linearization point, and to store the abstract output of the linearization step until it is need at the return step.

Let σ and σ' be status values, and λ be a list of parameters comprising gs, gs', ls and ls', and possibly in or out. For a step COP which is not a linearization step, the proof obligation is always of the following form.

$$\begin{aligned}
&\forall\, as : AS;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In;\ out : Out \bullet\\
&\qquad ABS(as, gs) \land INV(gs, ls) \land status(gs, ls) = \sigma \land COP(\lambda) \Rightarrow\\
&\qquad\qquad status(gs', ls') = \sigma' \land ABS(as, gs') \land INV(gs', ls')
\end{aligned} \tag{1}$$

For a linearization step such as the step $Push5t$ which simulates an abstract operation AOP, the proof obligation is of the following form.

$$\begin{aligned}
&\forall\, as : AS;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In \bullet\\
&\qquad ABS(as, gs) \land INV(gs, ls) \land status(gs, ls) = \sigma \land COP(\lambda) \Rightarrow\\
&\qquad\qquad (\exists\, as' : AS;\ out : Out \bullet AOP(in, as, as', out) \land\\
&\qquad\qquad\qquad status(gs', ls') = \sigma' \land ABS(as', gs') \land INV(gs', ls')
\end{aligned} \tag{2}$$

Step 2. Secondly, we need to prove non-interference between threads. This amounts to showing that a thread p (with local state ls) cannot invalidate the invariant $INV(gs, lsq)$ or change the status $status(gs, lsq)$ which another thread q (with local state lsq) relies on. To do this we require a further invariant $D(ls, lsq)$ relating the local states of two threads. For the Treiber stack, this invariant includes a predicate that the local variable n of two threads cannot be the same reference. That is, D includes the conjunct $pcq \in 2..5 \land pc \in 2..5 \Rightarrow n \neq nq$.

The proof obligation then requires we prove

$$\begin{aligned}
&\forall\, as : AS;\ gs, gs' : GS;\ ls, ls', lsq : LS \bullet\\
&\qquad ABS(as, gs) \land INV(gs, ls) \land INV(gs, lsq) \land D(ls, lsq) \land COP(\lambda)\\
&\qquad\qquad \Rightarrow INV(gs', lsq) \land D(ls', lsq) \land status(gs', lsq) = status(gs, lsq)
\end{aligned} \tag{3}$$

Additionally, we have a proof obligation related to initialisation.

$$\begin{aligned}
&\forall\, gs : GSInit \bullet \exists\, as : ASInit \bullet ABS(as, gs) \land\\
&\qquad (\forall\, ls : LSInit \bullet INV(gs, ls)) \land (\forall\, ls, lsq : LSInit \bullet D(ls, lsq))
\end{aligned} \tag{4}$$

Each of these proof obligations is step-local, involving a single line of code, and involving the states of at most two threads. Together they have been shown to prove linearizability between the abstract and concrete specifications [7].

3 Encoding the rules for deterministic specifications

To verify the Treiber stack, the theorem proving approach using KIV proposed in [7] requires 295 proof steps, 85 of which are interactive. If an error is found in either the implementation or the abstraction relation and invariants, all proof steps need to be redone once it is corrected. Our model checking approach can be applied to find such errors automatically before any proof steps are attempted. It uses the abstraction relation and invariants proposed for the proof steps in order to do this. KIV can then be applied to ensure no errors have been missed due to the limited state space used during the model checking.

In this section, we show how to encode each proof obligation as a separate model checking problem in TLC. We can alternatively encode a model checking problem that checks several, or even all, of the proof obligations at once. Separating the proof obligations, however, improves scalability; each model checking problem involves only one step of a single, generally deterministic operation.[4] Whether this separation needs to be done depends on the size of the state space of our abstract and concrete specifications.

A TLC model checking problem comprises a TLA+ module (encoding a specification in terms of variables, constants and definitions, including an initial-state and next-state definition), and a configuration file assigning finite values to constants, and listing the properties that need to be checked.

For each concrete step, we need to prove the proof obligations for the one or two operations derived from the line of code. Part of these obligations is that the *status* function is updated correctly. We assume we have already classified the type of each step, e.g., whether it is a linearization step or not. The remaining purpose of *status* is to store the input and output values until they are needed. To capture this we include *in* and *out* as variables, and introduce an invariant *STATUS* over them. For the Treiber stack, we have

$$\begin{aligned} \mathit{STATUS} \mathrel{\widehat{=}} & (pc \in 1..5 \Rightarrow in = v) \wedge (pc \in 8..9 \Rightarrow out = empty) \\ & \wedge (pc = 13 \Rightarrow out = lv) \end{aligned}$$

3.1 Non-linearization steps

For each concrete step which is not a linearization step, we need to prove proof obligation (1) for each operation *COP* derived from the line of code. As an example, consider the operation *Push*0. We need to show that when this operation occurs from a state satisfying $\mathit{ABS} \wedge \mathit{INV} \wedge \mathit{STATUS}$, it results in a state satisfying these conditions. Hence, we build a model which, from such a state, can do a single *Push*0 operation, and prove that *ABS*, *INV* and *STATUS* are invariants.[5]

The initial state space of such a model will be large, and hence we employ some simple strategies to reduce it. Firstly, we can ignore local variables that are not used in the `push` operation, i.e., the variables `ssn` and `lv` do not have to be included in the state. Similarly, since the operation has no output, the variable *out* can be left out of the state. Since these variables can occur in *INV* and *STATUS* we additionally remove all conjuncts of *INV* and *STATUS* that are not relevant immediately before or after the step, i.e., for *Push*0 all conjuncts that are not relevant when $pc = 0$ or $pc = 1$. Also, to reduce the size of the initial state, we can equate all local variables and *in* to default values (since they have not yet been assigned values at this step of `push`).

For *Push*0, the state is defined in terms of variables *stack*, *head*, *mem*, *n*, *ss*, *v*, *pc* and *in*. We also have constants *Ref*, *T* and *null*, as well as 2 additional

[4] An example of a nondeterministic operation is an invocation operation that takes an input. Such an operation is nondeterministic on the value of that input.

[5] The output of the model checker run can be checked to ensure that this model does not have an empty set of initial states.

constants N and *undef*. N is the maximum size of the stack, and *undef* is required since, unlike Z, TLA+ does not support partial functions; we model a partial function with a total function f by letting $f[e] = undef$ when e is not in the domain of the partial function. The initial state is then

$$
\begin{aligned}
Init \mathrel{\widehat{=}}\ & stack \in FiniteSeq(T, N) \\
& \wedge mem \in [Ref \rightarrow ((T \times (Ref \cup \{null\})) \cup undef)] \\
& \wedge head \in Ref \cup \{null\} \wedge ABS \wedge n = null \wedge ss = null \wedge v = 0 \\
& \wedge pc = 0 \wedge INV \wedge in = 0 \wedge STATUS
\end{aligned}
$$

where *FiniteSeq* is defined in terms of the function *Seq* of TLA+ [13]. Since TLC evaluates predicates from left to right it is necessary that all variables appearing in the definitions *ABS*, *INV* and *STATUS* are typed in a conjunct appearing to the left of them. Furthermore, since these definitions constrain the set of states under consideration, it is more efficient to have them as early as possible in the predicate, i.e., immediately following the typing of their variables.

The next-state relation is then defined in terms of the single operation

$$
\begin{aligned}
Push0 \mathrel{\widehat{=}}\ & pc = 0 \wedge pc' = 1 \wedge v' \in Ref \wedge in' = v' \\
& \wedge UNCHANGED\langle stack, head, mem, n, ss\rangle
\end{aligned}
$$

where *UNCHANGED* is a TLA+ operator for stating that particular variables are not changed by an operation.

The complete TLA+ module is shown below.[6]

———————————— MODULE *Push0* ————————————

EXTENDS *FiniteSequences*, *Naturals*

VARIABLES *stack*, *head*, *mem*, *n*, *ss*, *v*, *in*, *pc*

CONSTANTS *Ref*, *T*, *null*, *N*, *undef*

$$
\begin{aligned}
& ABS0[s \in FiniteSeq(T, N), h \in Ref \cup \{null\}] \mathrel{\widehat{=}} \\
& \quad (h = null \Rightarrow Lens(s) = 0) \\
& \quad \wedge (h \neq null \Rightarrow Len(s) \neq 0 \wedge mem[h] \neq undef \wedge mem[h][1] = Head(s) \\
& \qquad\qquad \wedge ABS0[Tail(S), mem[h][2]])
\end{aligned}
$$

$ABS \mathrel{\widehat{=}} ABS0[stack, head]$

$INV \mathrel{\widehat{=}} ...$ as described in the text

$STATUS \mathrel{\widehat{=}} pc \in 1..5 \Rightarrow in = v$

$Init \mathrel{\widehat{=}} ...$ as given above

$Push0 \mathrel{\widehat{=}} ...$ as given above

$Spec \mathrel{\widehat{=}} Init \wedge \Box[Push0]_{\langle stack, head, mem, n, ss, v, in, pc\rangle}$

————————————————————————————————

[6] The notation $Init \wedge \Box[Op]_{\langle v_1, \ldots, v_n\rangle}$ describes the module's behaviours whose initial states satisfy *Init* and whose state transitions satisfy *Op*, and specifies that the environment of the module is unable to change the values of $v_1, \ldots, v_n$.

Modules for other non-linearization steps are constructed similarly. For example, the module for operation $Push2$ has the same variables and constants. However, the initial state cannot assign n to $null$ as it is assigned a value in the previous line of code. Therefore, we have $n \in Ref$, rather than $n = null$ in $Init$; the other conjuncts of $Init$ being the same as before.

The next-state relation for $Push2$ is

$$\begin{aligned} Push2 \mathrel{\widehat{=}} & \ pc = 2 \wedge pc' = 3 \wedge mem' = [mem\ EXCEPT\ ![n] = \langle v, @[2] \rangle] \\ & \wedge UNCHANGED \langle stack, head, n, ss, v, in \rangle \end{aligned}$$

where the TLA+ notation $f' = [f\ \ EXCEPT\ ![n] = e]$ updates the function f so that $f'[n] = e$, where @ in e equals $f[n]$, e.g., $\langle v, @[2] \rangle = \langle v, mem[n][2] \rangle$ in $Push2$ above.

3.2 Linearization step

For each linearization step, we need to prove proof obligation (2). This proof obligation again requires that ABS, INV and $STATUS$ hold after the concrete step. However, the values for out and the abstract states after the step are values reached by applying the abstract operation AOP. To simplify the encoding, we assume two properties of the abstract operation in this section. We return to more general abstract operations in Section 4.

The first property is that abstract operations are deterministic. The second is that they are *total*, i.e., have a true guard and so can be applied at any time. Both of these properties are true of our specification of the Treiber stack in Section 2.1.

Proof obligation (2) is of the form

$$\forall x, y, y' \bullet P(x, y, y') \Rightarrow (\exists x' \bullet Q(x, x') \wedge R(x', y'))$$

where $Q(x, x')$ is the abstract operation. If this operation is deterministic, we have $Q(x, x') \equiv q(x) \wedge x' = e$ for some expression e and predicate $q(x)$. If it is also total then $q(x) = true$ and we have $Q(x, x') \equiv x' = e$. Therefore, proof obligation (2) can be written as

$$\forall x, y, y' \bullet P(x, y, y') \Rightarrow (\exists x' \bullet x' = e \wedge R(x', y'))$$

Applying the one-point rule for existential quantification ($\exists x \bullet x = e \wedge P(x) \equiv P(e)$), to $\exists x' \bullet x' = e \wedge R(x', y')$ we get

$$\forall x, y, y' \bullet P(x, y, y') \Rightarrow R(e, y')$$

Then, applying the one-point rule for universal quantification ($P(e) \Rightarrow R(e) \equiv \forall x \bullet P(x) \wedge x = e \Rightarrow R(x)$), to $P(x, y, y') \Rightarrow R(e, y')$ we get

$$\forall x, x', y, y' \bullet P(x, y, y') \wedge x' = e \Rightarrow R(x', y')$$

which is

$$\forall\, x, x', y, y' \bullet P(x, y, y') \wedge Q(x, x') \Rightarrow R(x', y')$$

Hence, we can prove proof obligation (2) in the same way we prove proof obligation (1) after extending the next-state relation to produce the unique values for the abstract specification. For example, for $Push5t$ we have

$$\begin{aligned} Push5t \mathrel{\widehat{=}}\ & pc = 5 \wedge head = ss \wedge pc' = 6 \wedge head' = n \\ & \wedge stack' = \langle in \rangle \circ stack \wedge UNCHANGED\langle mem, n, ss, v, in \rangle \end{aligned}$$

where $s \circ t$ concatenates sequences s and t. That is, $stack$ is updated according to the abstract operation $Push$ of Section 2.1.

3.3 Non-interference

For each concrete step, whether a linearization step or not, we need to prove proof obligation (3). This proof obligation requires that under an invariant D the actions of one thread p do not break the invariant INV of another thread q. Again, for scalability, we decide to encode the proof obligation for a single step of p and for a single state of q. For example, we will have one TLA+ module for the case when p executes $Push5t$ while q is at line 2.

To encode such a module we need to have local variables for both p and q and invariants $INVq$ and $STATUSq$ for q, as well as the new invariant D. The module is as follows.

MODULE $Push5tPush2$

EXTENDS $FiniteSequences, Naturals$

VARIABLES $stack, head, mem, n, ss, v, in, pc, nq, ssq, vq, inq, pcq$

CONSTANTS $Ref, T, null, N, undef$

$ABS \mathrel{\widehat{=}} ...$ as before

$INV \mathrel{\widehat{=}} ...$ as before

$INVq \mathrel{\widehat{=}} ...$ like INV but in terms of nq, etc.

$D \mathrel{\widehat{=}} ...$ as described in the text

$STATUS \mathrel{\widehat{=}} ...$ as before

$STATUSq \mathrel{\widehat{=}} ...$ like STATUS but in terms of nq, etc.

$$\begin{aligned} Init \mathrel{\widehat{=}}\ & stack \in FiniteSeq(T, N) \\ & \wedge mem \in [Ref \rightarrow ((T \times (Ref \cup \{null\})) \cup undef)] \\ & \wedge head \in Ref \cup \{null\} \wedge ABS \wedge n \in Ref \wedge ss \in Ref \cup \{null\} \\ & \wedge v \in T \wedge pc = 5 \wedge INV \wedge in \in T \wedge STATUS \wedge nq \in Ref \\ & \wedge ssq = null \wedge vq \in T \wedge pcq = 2 \wedge INVq \wedge D \wedge inq \in T \\ & \wedge STATUSq \end{aligned}$$

$Push5t \mathrel{\widehat{=}} ...$ as given above except nq, etc. in the $UNCHANGED$ list

$Spec \mathrel{\widehat{=}} Init \wedge \Box[Push5t]_{\langle stack, head, mem, n, ss, v, in, pc, nq, ssq, vq, inq, pcq \rangle}$

Given this module, we then prove $INVq$, D and $STATUSq$ are invariants to discharge proof obligation (3).

3.4 Initialisation

The final proof obligation (4) is proved by creating a module with 2 local states (as above). All variables, global and local, are initialised according to the abstract and concrete initialisation schemas, or in the case of local variables, given a default value (since they are not assigned a value initially when the threads are idle). Then we check that ABS, INV, $INVq$ and D are invariants under the empty next-state relation. That is, the required module is

———— MODULE $Init$ ————

EXTENDS $FiniteSequences, Naturals$

VARIABLES $stack, head, mem, n, ss, ssn, v, lv, pc, nq, ssq, ssnq, vq, lvq, pcq$

CONSTANTS $Ref, T, null, N, undef$

$ABS \mathrel{\widehat{=}} \ldots$ as before

$INV \mathrel{\widehat{=}} \ldots$ as before

$INVq \mathrel{\widehat{=}} \ldots$ like INV but in terms of nq, etc.

$D \mathrel{\widehat{=}} \ldots$ as described in the text

$$
\begin{aligned}
Init \mathrel{\widehat{=}}\ & stack = \{\} \land mem \in [Ref \rightarrow \{undef\}] \land head = null \\
& \land n = null \land ss = null \land ssn = null \land v = 0 \land lv = 0 \land pc = 0 \\
& \land nq = null \land ssq = null \land ssnq = null \land vq = 0 \land lvq = 0 \\
& \land pcq = 0
\end{aligned}
$$

$Stop \mathrel{\widehat{=}} FALSE$

$Spec \mathrel{\widehat{=}} Init \land \Box[Stop]_{\langle stack,head,mem,n,ss,ssn,v,lv,pc,nq,ssq,ssnq,vq,lvq,pcq \rangle}$

3.5 Discussion

To make our approach practical we need to address the fact that it requires many model checking jobs to be run. A batch program is required to handle these jobs and report any errors that are encountered. While developing our approach, we ran the jobs manually[7] and were able to successfully verify the Treiber stack, a test-and-test-and-set spinlock implementation taken from [10] and an implementation of the Linux reader-writer mechanism, seqlock, taken from [3].

Checking a single proof obligation for the Treiber stack with a maximum stack size of 4 takes around 16 seconds on an iMac with a 2.7GHz Intel Core

[7] To save time, we often ran multiple jobs at once, i.e., using one module, at the expense of a smaller state space.

i5 processor and 4GB RAM. Since it is intended that the full verification of the stack is to be carried out using KIV, this stack size is sufficient for our purposes.

In general, however, checking larger state spaces has the potential to uncover more errors. One area of future work is to look at improving the efficiency of our approach. Although TLC is capable of running multiple threads, these are only employed after the initial states have been computed [27]. Hence, reducing the number of initial states by ignoring unused local variables, and setting local variables which have not been assigned a value to a default value is important. Since we can determine when to apply these state space reductions statically, this process can be automated. Using a different encoding where the initial state is built up over a number of state transitions would enable us to use TLC's option to run a user-defined number of threads. Then efficiency could be improved by using more and better hardware. For example, Amazon run TLC on a cluster of 10 machines, each with eight cores plus hyperthreads and 23GB of RAM [16].

Another area of future work is to investigate encoding the simulation rules in other model checkers such as SAL [5] to compare efficiency.

4 Encoding the rules for nondeterministic specifications

Consider a bounded version of the Treiber stack whose specification abstracts from what happens when a push occurs and the stack is full. The state schema and operation *Push* are updated as follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{AS} \\
\hline
stack : seq T \\
\hline
\#stack \leq Max \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Push} \\
\hline
\Delta AS \\
v? : T \\
\hline
\#stack < Max \Rightarrow \\
\quad stack = \langle v? \rangle \frown stack \\
\hline
\end{array}$$

We could implement *Push* to simply ignore the new value when the stack is full. Alternatively, we could implement it to delete the oldest value in the stack, in order to make place for the new value. Whether such implementations are sensible would depend on the envisaged application.

To prove any such implementation is linearizable with respect to *Push*, we cannot use the approach of Section 3.2 which relies on *Push* being deterministic. Instead we encode proof obligation (2) more directly. Instead of proving that ABS is an invariant for all concrete steps, we instead prove that for the linearization step of a nondeterministic operation that there exists an execution of the abstract operation which leads to ABS being true. That is, for the module corresponding to step $Push5t$ we prove $ABS1$ is an invariant, where

$$\begin{aligned}
ABS1 \mathrel{\widehat{=}} {} & (pc = 5 \Rightarrow ABS) \\
& \wedge (pc = 6 \Rightarrow (\exists\, s \in FiniteSeq(T, N) \bullet \\
& \qquad Len(stack) < Max \Rightarrow s = (\langle in \rangle \circ stack) \wedge ABS0[s, head]))
\end{aligned}$$

and *Push*5*t* is encoded in the same way as a non-linearization step. The model checking time for this encoding is comparable to that of Section 4, taking around 16 seconds for a stack of maximum size 4.

A similar approach can be used for an abstract operation which is not total, ensuring the linearization step occurs only when the operation is enabled.

5 Conclusion

In this paper, we have provided model checking support for a simulation-based approach to proving linearizability [7]. The approach enables developers of concurrent objects to quickly check their designs for errors before attempting a full verification using a theorem prover. The approach is the only model checking approach we are aware of that allows checking linearizability for an arbitrary number of threads. Other approaches are typically limited to between 2 and 4 threads depending on the concurrent object.

At present, the approach can only be used with concurrent objects whose linearization points can be determined from the current state of the calling thread and object. As future work, we would like to extend this to other concurrent objects. As a first step, we will investigate encoding the additional simulation rules of [8], allowing objects whose linearization points are determined by future states. These simulation rules are only slightly more complicated than the ones we encoded in this paper. Following this, we will investigate handling the complete approach, for all possible concurrent objects, described in [18]. This approach requires that the implementation is a backward simulation of the specification. Earlier work on verifying backward simulations using model checking [19, 20] will provide a starting point for this investigation.

Acknowledgements Thanks to Kirsten Winter for her helpful comments. This work was supported by ARC Discovery Grant DP160102457.

References

1. D. Amit, N. Rinetzky, T.W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV 2007*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.
2. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A complete and automatic linearizability checker. In *PLDI '10*, pages 330–340. ACM, 2010.
3. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP 2012*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.
4. C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS 2007*, volume 4634 of *LNCS*, pages 233–238. Springer, 2007.
5. L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *CAV 2004*, volume 3114 of *LNCS*, pages 496–500. Springer-Verlag, 2004.
6. J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* Springer, 2nd edition, 2014.

7. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.
8. J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisabilty with potential linearisation points. In *FM 2011*, volume 6664 of *LNCS*, pages 323–337. Springer, 2011.
9. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE 2004*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.
10. A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In *DISC 2012*, volume 7611 of *LNCS*, pages 31–45. Springer, 2012.
11. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
12. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
13. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman, 2002.
14. Y. Liu, W. Chen, Y.A. Liu, and J. Sun. Model checking linearizability via refinement. In *FM '09*, volume 5850 of *LNCS*, pages 321–337. Springer, 2009.
15. C. Newcombe. Why Amazon chose TLA+. In *ABZ 2014*, volume 8477 of *LNCS*, pages 25–39. Springer, 2014.
16. C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.
17. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In *Automated Deduction*, pages 13–39. Kluwer, 1998.
18. G. Schellhorn, H. Wehrheim, and J. Derrick. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. on Computational Logic*, 2014.
19. G. Smith and J. Derrick. Verifying data refinements using a model checker. *Formal Aspects of Computing*, 18(3):264–287, 2006.
20. G. Smith and K. Winter. Model checking action system refinements. *Formal Aspects of Computing*, 21(1-2):155–186, 2009.
21. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.
22. O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In *HVC2013*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013.
23. R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Res. Ctr., 1986.
24. V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.
25. P. Černý, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent list implementations. In *CAV'10*, volume 6174 of *LNCS*, pages 465–479. Springer, 2010.
26. M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *SPIN 2009*, volume 5578 of *LNCS*, pages 261–278. Springer, 2009.
27. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *CHARME99*, volume 1703 of *LNCS*, pages 54–66. Springer, 1999.
28. S.J. Zhang. Scalable automatic linearizability checking. In *ICSE '11*, pages 1185–1187. ACM, 2011.