
Proceedings Paper:
Smith, G. and Derrick, J. (2016) Invariant generation for linearizability proofs. In: Proceedings of the ACM Symposium on Applied Computing. 31st Annual ACM Symposium on Applied Computing, 04/04/2016 - 08/04/2016, Pisa, Italy. http://doi.org/10.1145/2851613.2851837. ACM, pp. 1694-1699. ISBN: 9781450337397.

https://doi.org/10.1145/2851613.2851837

Invariant generation for linearizability proofs

Graeme Smith
School of Information Technology and Electrical Engineering
The University of Queensland, Australia
smith@itee.uq.edu.au

John Derrick
Department of Computing
University of Sheffield, UK
j.derrick@sheffield.ac.uk

ABSTRACT

Linearizability is a widely recognised correctness criterion for concurrent objects. A number of proof methods for verifying linearizability exist. In this paper, we simplify one such method with a systematic approach for invariant generation. Although this existing refinement-based method is itself systematic and fully tool-supported, it requires the verifier to provide a specific invariant over the implementation. While a chosen invariant may suffice for some proof obligations of the method, it may not for others resulting in a new, stronger invariant to be chosen and the previously completed proof steps to be redone. Our approach avoids such wasted proof effort by generating an invariant which is guaranteed to be sufficient for all proof obligations.

CCS Concepts

•Software and its engineering → Software verification;

Keywords

Correctness proofs; Formal Methods; Specifying and Verifying and Reasoning about Programs; Linearizability; Invariant generation.

1. INTRODUCTION

Linearizability [9] is widely regarded as the standard correctness criterion for concurrent objects. Given an abstract specification and a proposed implementation, linearizability requires the existence of a sequential execution of the abstract specification for every concurrent execution of the implementation such that the abstract and concrete executions produce the same behaviour. The sequential execution is obtained by identifying *linearization points* at which the potentially overlapping concurrent operations are deemed to take effect instantaneously.

A variety of methods for verifying linearizabilty have been developed. These range from using shape analysis [1, 2] and separation logic [2] to rely-guarantee reasoning [13] and refinement-based methods [7, 4, 11]. In particular, Derrick et al. have developed a refinement-based method for verifying linearizability [3, 4, 5, 11]. This approach is fully encoded in a theorem proving tool, KIV [10], and has been proved sound and complete — the proofs themselves being done within KIV. Its completeness relies on the fact that the refinement-based approach allows backwards simulation [11]. This is necessary when the linearization point can only be determined by examining the full global history of the concurrent object (for example, as in the queue implementation of Herlihy and Wing [9] and the elimination stack of Hendler et al. [8]).

A strength of this method is that it allows the verifier to reason about single concrete steps (i.e., lines of code) in isolation, showing that they simulate either an abstract skip or, at a linearization point, an abstract operation. This simplifies the proof of linearizability immensely; while there may be many simulation proofs to perform, each is with respect to a single line of code which is not more complicated than an assignment or change of program counter.

However, the method of Derrick et al. requires the verifier to provide both an abstraction relation relating the concrete and abstract specifications, and an invariant on the concrete state enabling the proof to be carried out by simulation. Finding the latter is non-trivial. In practice, it requires an iterative process in which the invariant needs to be repeatedly strengthened as the proofs for different lines of code are performed. When proving linearizability on weak memory models, such as TSO [6], the process is further complicated by the large number of different states that can exist at any line of code due to the use of store buffers potentially delaying the commitment of previous write instructions [12].

In this paper, we provide a systematic approach to generating the required invariant for a linearizability proof. We begin in Section 2 by discussing the method of Derrick et al. in more detail. In Section 3 we present the basic approach to invariant generation and in Section 4 illustrate it on the Linux reader/writer mechanism *seqlock*. We discuss extending the approach to cover a wider range of concurrent objects in Section 5.

2. BACKGROUND

Consider the Linux reader/writer mechanism *seqlock*, that allows reading of shared variables without locking the global memory, thus supporting fast write access. A process wishing to *write* to the shared variables $x1$ and $x2$ *acquires* a software lock and increments a counter c. It then proceeds

SAC 2016, April 04-08, 2016, Pisa, Italy

DOI: http://dx.doi.org/10.1145/2851613.2851837

```
int x1 = 0, x2 = 0;

write(int d1,d2) {          read() {
   x1 = d1;                       return(x1,x2);
   x2 = d2;                 }
}
```

Figure 1: *seqlock* **specification**

```
int x1 = 0, x2 = 0;          read() {
int c = 0;                        int c0, d1, d2;
                                  do {
write(int d1,d2) {                 do {
 1  acquire;                       7      c0 = c;
 2  c++;                           8    } while(c0%2 != 0);
 3  x1 = d1;                       9    d1 = x1;
 4  x2 = d2;                       10   d2 = x2;
 5  c++;                           11 } while(c != c0);
 6  release;                       12 return(d1,d2);
}                              }
```

Figure 2: *seqlock* **implementation**

to write to the variables, and finally increments c again before *releasing* the lock. The lock ensures synchronisation between writers, and the counter c ensures the consistency of values read by other processes. The two increments of c ensure that it is odd when a process is writing to the variables, and even otherwise. Hence, when a process wishes to *read* the shared variables, it waits in a loop until c is even before reading them. Also, before returning it checks that the value of c has not changed (i.e., another write has not begun). If it has changed, the process starts over.

An abstract specification of *seqlock* is given in Figure 1. A typical implementation (with line numbers from 1 to 12) is given in Figure 2. In the implementation, a local variable $c0$ is used by the *read* operation to record the (even) value of c before the operation begins updating the shared variables.

Correctness requires showing all concrete histories are linearizable. Following the approach of [4], we have two proof steps 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 *write* we require that line 5, the second $c{+}{+}$, simulates the abstract operation and all other lines simulate an abstract skip (see Figure 3 for an execution of the operation).

To do this we need to define an abstraction relation relating the global (i.e., shared) concrete state space gs and abstract state space as. The abstraction relation $ABS(as, gs)$ for *seqlock* is determined in a straightforward manner from the understanding of c's role in the algorithm. That is, whenever c is even, the abstract and concrete values of $x1$ and $x2$ are equal. Hence $ABS(as, gs)$ is defined as

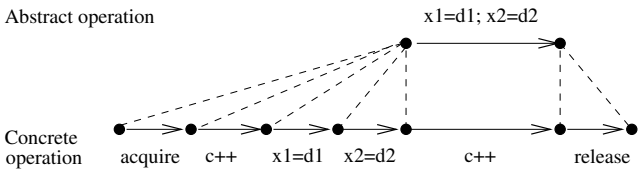

Figure 3: Simulation of *Write*

$$gs.c \bmod 2 = 0 \Rightarrow gs.x1 = as.x1 \wedge gs.x2 = as.x2$$

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 line 5 of the code simulates the effect of the abstract operation, this invariant needs to ensure that at line 5 we have: $c \bmod 2 \neq 0 \Rightarrow x1 = d1 \wedge x2 = d2$. Such an invariant is stated in terms of the global concrete state space gs and a local concrete state space ls comprising a program counter variable pc (set to 0 when no operation is executing) and the local variables used in operations, e.g., $c0$, $d1$ and $d2$ in *seqlock*. Hence, the invariant $INV(gs, ls)$ must imply the following.

$$ls.pc = 5 \Rightarrow (gs.c \bmod 2 = 0 \Rightarrow gs.x1 = d1 \wedge gs.x2 = d2)$$

Each simulation is then proved by one of 5 rules depending on whether the line of code is an invocation (first line of an operation), return (last line of an operation) or internal step (neither an invocation nor return), and whether it occurs before or after the linearization step (see Figure 4(a)). Each rule concerns a single line of code (whose semantics is captured by COP) and in the case of linearization steps, an abstract operation (whose semantics is captured by AOP). 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.

Step 2. Secondly, we need to prove non-interference between threads. This amounts to showing that a thread p running the concrete code cannot, by changing the global concrete state space, invalidate the invariant which another thread q (running the same code) relies on. For example, a thread p should not be able to change the value of $x1$ when a thread q is at line 5 since this would invalidate the requirement on $INV(gs, ls)$ above. To do this we require a further invariant $D(ls, lsq)$ relating the local states of two threads whose local states are ls and lsq (see Figure 4(b)). This invariant is normally easier to derive, usually stating that certain lines of code are mutually exclusive, or that local objects are not aliased. It is not generated by our approach.

Additionally, we have a proof step related to initialisation (see Figure 4(c)).

Each of the proof obligations relies on the implementation invariant $INV(gs, ls)$. Determining the invariant is not always straightforward. Developing it iteratively by strengthening it as required for each simulation leads to a lot of extra work. Each time the invariant is strengthened, all previously completed simulations need to be re-performed

(a) Simulation.

(i) Invocation (the invocation may be the linearization step or simulate an abstract skip).

$$\begin{array}{l}\forall\, as : AS;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In \bullet \\ \quad R(as, gs, ls) \wedge status(gs, ls) = IDLE \wedge COP(in, gs, ls, gs', ls') \Rightarrow \\ \quad\quad status(gs', ls') = IN(in) \wedge R(as, gs', ls') \\ \quad\quad \vee \\ \quad\quad (\exists\, as' : AS;\ out : Out \bullet \\ \quad\quad\quad AOP(in, as, as', out) \wedge status(gs', ls') = OUT(out) \wedge R(as', gs', ls'))\end{array}$$

(ii) Before Linearization (the internal line of code may be the linearization step or simulate an abstract skip).

$$\begin{array}{l}\forall\, as : AS;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In \bullet \\ \quad R(as, gs, ls) \wedge status(gs, ls) = IN(in) \wedge COP(gs, ls, gs', ls') \Rightarrow \\ \quad\quad status(gs', ls') = IN(in) \wedge R(as, gs', ls') \\ \quad\quad \vee \\ \quad\quad (\exists\, as' : AS;\ out : Out \bullet \\ \quad\quad\quad AOP(in, as, as', out) \wedge status(gs', ls') = OUT(out) \wedge R(as', gs', ls'))\end{array}$$

(iii) After Linearization (the internal line of code simulates an abstract skip).

$$\begin{array}{l}\forall\, as : AS;\ gs, gs' : GS;\ ls, ls' : LS;\ out : Out \bullet \\ \quad R(as, gs, ls) \wedge status(gs, ls) = OUT(out) \wedge COP(gs, ls, gs', ls') \Rightarrow \\ \quad\quad status(gs', ls') = OUT(out) \wedge R(as, gs', ls')\end{array}$$

(iv) Return before Linearization (the return is the linearization step).

$$\begin{array}{l}\forall\, as : AS;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In \bullet \\ \quad R(as, gs, ls) \wedge status(gs, ls) = IN(in) \wedge COP(gs, ls, gs', ls', out) \Rightarrow \\ \quad\quad\quad status(gs', ls') = IDLE \wedge \\ \quad\quad\quad (\exists\, as' : AS;\ out : Out \bullet AOP(in, as, as', out) \wedge R(as', gs', ls'))\end{array}$$

(v) Return after Linearization (the return simuates an abstract skip operation).

$$\begin{array}{l}\forall\, as : AS;\ gs, gs' : GS;\ ls, ls' : LS;\ out : Out \bullet \\ \quad R(as, gs, ls) \wedge status(gs, ls) = OUT(out) \wedge COP(gs, ls, gs', ls', out') \Rightarrow \\ \quad\quad\quad out' = out \wedge status(gs', ls') = IDLE \wedge R(as, gs', ls'))))\end{array}$$

(b) Non-interference (where λ includes gs, gs', ls and ls' and possibly in or out depending on COP).

$$\begin{array}{l}\forall\, as : AS;\ gs, gs' : GS;\ ls, ls', lsq : LS \bullet \\ \quad ABS(as, gs) \wedge INV(gs, ls) \wedge INV(gs, lsq) \wedge D(ls, lsq) \wedge COP(\lambda) \\ \quad\quad \Rightarrow INV(gs', lsq) \wedge D(ls', lsq) \wedge status(gs', lsq) = status(gs, lsq)\end{array}$$

(c) Initialisation.

$$\begin{array}{l}\forall\, gs : GSInit \bullet \exists\, as : AInit \bullet \\ \quad ABS(as, gs) \wedge (\forall\, ls : LSInit \bullet INV(gs, ls)) \wedge (\forall\, ls, lsq : LSInit \bullet D(ls, lsq))\end{array}$$

Figure 4: Proof obligations from [4]. AS, GS **and** LS **are the types of the abstract states, global concrete states and local concrete states respectively.** $AInit$, $GSInit$ **and** $LSInit$ **are the restrictions of these types to initial states. Primed states, e.g.,** gs'**, represent post-states of operations whereas unprimed states, e.g.,** gs**, represent pre-states.** $R(as, gs, ls) = ABS(as, gs) \wedge INV(gs, ls)$**.** $ABS(as, gs)$**,** $INV(gs, ls)$ **and** $D(ls, lsq)$ **are relations between states as described in the text.**

with the new invariant. By systematically generating the invariant to ensure the simulation proofs hold, this proof effort can be greatly reduced.

3. INVARIANT GENERATION

The invariant we generate is a conjunction of predicates of the form $(ls.pc = n \Rightarrow I_n(gs, ls))$ where n is a line number. The key to generating it is in recognising that its role is to ensure the required simulation holds at each program step. That is, at the line number n we require a certain condition to be true, and the lines preceding n must ensure this condition becomes true.

We will assume that there is exactly one return step and associated linearization step per operation. We will come back to this assumption at the end of this section. To generate the required invariant we start with the final step of an operation execution and work backwards through the steps of the execution as follows.

For a linearization step (such as the second $c{+}{+}$ in Figure 3), the proof obligation from Figure 4(a) is always of the following form (where σ and σ' are status values and λ is a list of parameters)

$$\begin{array}{l}\forall\, as : AS;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In \bullet \\ \quad R(as, gs, ls) \wedge status(gs, ls) = \sigma \wedge COP(\lambda) \Rightarrow \\ \quad\quad (\exists\, as' : AS;\ out : Out \bullet \\ \quad\quad\quad AOP(in, as, as', out) \wedge \\ \quad\quad\quad status(gs', ls') = \sigma' \wedge R(as', gs', ls')\end{array} \quad (1)$$

irrespective of whether the step is an invocation, return or internal step.

Let such a step end at line m. In the approach of Derrick et al. each step is deterministic (since branches in code are modelled as two separate steps), so there will be only one such line m. For the final step of an operation, m will be 0 denoting the idle state. A *starting invariant* $I_0(gs, ls)$ (true when $ls.pc = 0$) needs to be supplied by the verifier based on their understanding of the algorithm.

Theorem 1 Let COP be the linearization step of an abstract operation AOP which results in $ls.pc = m$ where $ls.pc = m \Rightarrow (INV(gs, ls) \Leftrightarrow I_m(gs, ls))$. Condition (1) will hold if the invariant has the following conjunct.

$$\begin{array}{l}\forall\, as : AS;\ gs' : GS;\ ls' : LS;\ in : In \bullet \\ \quad ABS(as, gs) \wedge status(gs, ls) = \sigma \wedge COP(\lambda) \Rightarrow \\ \quad\quad (\exists\, as' : AS;\ out : Out \bullet \\ \quad\quad\quad AOP(in, as, as', out) \wedge status(gs', ls') = \sigma' \\ \quad\quad\quad I_m(gs', ls') \wedge ABS(as', gs'))\end{array} \quad (2)$$

Proof Figure 5 shows that the consequent of (1) can be derived from its antecedent when the invariant includes the conjunct (2) and given that $ls.pc = m \Rightarrow (INV(gs, ls) \Leftrightarrow I_m(gs, ls))$. □

For a step which is not a linearization step of an operation execution, the proof obligation from Figure 4(a) is always of the following form (where σ and σ' are status values and λ is a list of parameters)

$$\begin{array}{l}\forall\, as : AS;\ gs, gs' : GS;\ ls, ls' : LS;\ in : In;\ out : Out \bullet \\ \quad R(as, gs, ls) \wedge status(gs, ls) = \sigma \wedge COP(\lambda) \Rightarrow \\ \quad\quad status(gs', ls') = \sigma' \wedge R(as, gs', ls')\end{array} \quad (3)$$

irrespective of whether the step is an invocation, return or internal step.

Theorem 2 Let COP be a step which results in $ls.pc = m$ such that $ls.pc = m \Rightarrow (INV(gs, ls) \Leftrightarrow I_m(gs, ls))$. Condition (3) will hold if the invariant has the following conjunct.

$$\begin{array}{l}\forall\, as : AS;\ gs' : GS;\ ls' : LS;\ in : In;\ out : Out \bullet \\ \quad ABS(as, gs) \wedge status(gs, ls) = \sigma \wedge COP(\lambda) \Rightarrow \\ \quad\quad status(gs', ls') = \sigma' \wedge I_m(gs', ls') \wedge \\ \quad\quad ABS(as, gs')\end{array} \quad (4)$$

Proof Figure 6 shows that the consequent of (3) can be derived from its antecedent when the invariant includes the conjunct (4) and given that $ls.pc = m \Rightarrow (INV(gs, ls) \Leftrightarrow I_m(gs, ls))$. □

It is important to note that (2) and (4) are the *weakest* conjuncts that will ensure (1) and (3) respectively. If we were to weaken either (2) or (4) by adding a conjunct to their antecedent which does not imply the existing antecedent, then the corresponding theorem would no longer hold.

This enables us to also use the approach when an operation does not have a unique return step and associated linearization step. The approach can be applied for each return step and the resulting conjuncts conjoined. Since the invariants are the weakest required to ensure the required simulations hold, if this conjunction leads to a contradiction then there is no invariant which can assure linearization for all returns, and hence the concrete algorithm is not linearizable.

4. CASE STUDY

The starting invariant for the *seqlock* case study is derived by considering the relationship between the global variables $x1$, $x2$ and c, and *lock*, a Boolean variable which is true precisely when the lock of the *write* operation is held. The values of the local variables are not significant when $ls.pc = 0$, so these need not be considered.

Since $x1$ and $x2$ can take on any values, the invariant involves only c and *lock*. It is clear from the code that c is even whenever the lock is not held (it is even initially and is incremented twice between each *acquire* and *release*). When the lock is held, c may be even or odd. Hence, $I_0(gs, ls) \mathrel{\widehat{=}} \neg\, gs.lock \Rightarrow gs.c \bmod 2 = 0$.

Since the linearization step of the *seqlock* operation *write* occurs when $ls.pc = 5$, we choose $status(gs, ls)$ such that $ls.pc = 0 \Rightarrow status(gs, ls) = IDLE$ and $ls.pc \in 1..5 \Rightarrow status(gs, ls) = IN(in)$ and $ls.pc = 6 \Rightarrow status(gs, ls) = OUT(\bot)$ where in is the tuple comprising the values of d_1 and d_2.

For the final step (*release*) of *write* to simulate an abstract skip operation, the required conjunct of the invariant is the following instantiation of (4).

$$\begin{array}{l}\forall\, as : AS;\ gs' : GS;\ ls' : LS;\ in : In;\ out : Out \bullet \\ \quad (gs.c \bmod 2 = 0 \Rightarrow gs.x1 = as.x1 \wedge gs.x2 = as.x2) \wedge \\ \quad OUT(\bot) = OUT(out) \wedge ls.pc = 6 \wedge ls'.pc = 0 \wedge \\ \quad \neg\, gs'.lock \wedge gs'.c = gs.c \wedge gs'.x1 = gs.x1 \wedge \\ \quad gs'.x2 = gs.x2 \Rightarrow \\ \quad\quad IDLE = IDLE \wedge (\neg\, gs'.lock \Rightarrow gs'.c \bmod 2 = 0) \wedge \\ \quad\quad (gs'.c \bmod 2 = 0 \Rightarrow \\ \quad\quad\quad gs'.x1 = as.x1 \wedge gs'.x2 = as.x2)\end{array}$$

Applying the one-point rule $(\forall\, x : T \bullet x = v \wedge P(x) \Rightarrow Q(x) \equiv P(v) \Rightarrow Q(v))$ to the quantified variables gs', ls', in and out, we get

1. $R(as, gs, ls)$ [antecedent of (1)]
2. $status(gs, ls) = \sigma$ [antecedent of (1)]
3. $COP(\lambda)$ [antecedent of (1)]
4. $ABS(as, gs)$ [1 and $R(as, gs, ls) = ABS(as, gs) \wedge INV(gs, ls)$]
5. $INV(gs, ls)$ [1 and $R(as, gs, ls) = ABS(as, gs) \wedge INV(gs, ls)$]
6. $\forall\, as : AS;\ gs' : GS;\ ls' : LS;\ in : In \bullet$
   $ABS(as, gs) \wedge status(gs, ls) = \sigma \wedge COP(\lambda) \Rightarrow$
   $(\exists\, as' : AS;\ out : Out \bullet AOP(in, as, as', out) \wedge status(gs', ls') = \sigma' \wedge I_m(gs', ls') \wedge ABS(as', gs'))$ [5,(2)]
7. $\exists\, as' : AS;\ out : Out \bullet$
   $AOP(in, as, as', out) \wedge status(gs', ls') = \sigma' \wedge I_m(gs', ls') \wedge ABS(as', gs')$ [2,3,4,6]
8. $ls'.pc = m$ [3, $COP(\lambda)$ ends at line m]
9. $\exists\, as' : AS;\ out : Out \bullet$
   $AOP(in, as, as', out) \wedge status(gs', ls') = \sigma' \wedge R(as', gs', ls')$ [7, 8 and $ls.pc = m \Rightarrow (INV(gs, ls) \Leftrightarrow I_m(gs, ls))$]

Figure 5: Proof of Theorem 1

1. $R(as, gs, ls)$ [antecedent of (3)]
2. $status(gs, ls) = \sigma$ [antecedent of (3)]
3. $COP(\lambda)$ [antecedent of (3)]
4. $ABS(as, gs)$ [1 and $R(as, gs, ls) = ABS(as, gs) \wedge INV(gs, ls)$]
5. $INV(gs, ls)$ [1 and $R(as, gs, ls) = ABS(as, gs) \wedge INV(gs, ls)$]
6. $\forall\, as : AS;\ gs' : GS;\ ls' : LS;\ in : In;\ out : Out \bullet$
   $ABS(as, gs) \wedge status(gs, ls) = \sigma \wedge COP(\lambda) \Rightarrow status(gs', ls') = \sigma' \wedge I_m(gs', ls') \wedge ABS(as, gs')$ [5,(4)]
7. $status(gs', ls') = \sigma' \wedge I_m(gs', ls') \wedge ABS(as, gs')$ [2,3,4,6]
8. $ls'.pc = m$ [3, $COP(\lambda)$ ends at line m]
9. $status(gs', ls') = \sigma' \wedge R(as', gs', ls')$ [7,8 and $ls.pc = m \Rightarrow (INV(gs, ls) \Leftrightarrow I_m(gs, ls))$]

Figure 6: Proof of Theorem 2

$$\forall\, as : AS \bullet$$
$$(gs.c \bmod 2 = 0 \Rightarrow gs.x1 = as.x1 \wedge gs.x2 = as.x2) \wedge$$
$$ls.pc = 6 \Rightarrow$$
$$gs.c \bmod 2 = 0 \wedge$$
$$(gs.c \bmod 2 = 0 \Rightarrow gs.x1 = as.x1 \wedge gs.x2 = as.x2)$$

which (using $A \wedge B \Rightarrow C \wedge A \ \equiv\ A \wedge B \Rightarrow C$)) simplifies to

$$\forall\, as : AS \bullet$$
$$(gs.c \bmod 2 = 0 \Rightarrow gs.x1 = as.x1 \wedge gs.x2 = as.x2) \wedge$$
$$ls.pc = 6 \Rightarrow$$
$$gs.c \bmod 2 = 0$$

which (since there is an $as : AS$ which satisfies the first line of the antecedent) simplifies to

$$ls.pc = 6 \Rightarrow gs.c \bmod 2 = 0\,.$$

That is, $I_6(gs, ls)$ is $gs.c \bmod 2 = 0$ and we need to ensure that the step that leads to $ls.pc = 6$ results in $I_6(gs, ls)$. Using this result, we can derive the invariant required by the preceding step (c++). Since this is the linearization step, the required conjunct of the invariant is the following instantiation of (2).

$$\forall\, as : AS;\ gs' : GS;\ ls' : LS;\ in : In \bullet$$
$$(gs.c \bmod 2 = 0 \Rightarrow gs.x1 = as.x1 \wedge gs.x2 = as.x2) \wedge$$
$$IN(d1, d2) = IN(in) \wedge ls.pc = 5 \wedge ls'.pc = 6 \wedge$$
$$gs'.c = gs.c + 1 \wedge gs'.x1 = gs.x1 \wedge gs'.x2 = gs.d2$$
$$gs'.lock = gs.lock \Rightarrow$$
$$(\exists\, as' : AS;\ out : Out \bullet$$
$$as'.x1 = d1 \wedge as'.x2 = d2 \wedge$$
$$OUT(\perp) = OUT(out) \wedge gs'.c \bmod 2 = 0 \wedge$$
$$(gs'.c \bmod 2 = 0 \Rightarrow$$
$$gs'.x1 = as'.x1 \wedge gs'.x2 = as'.x2))$$

Applying the one-point rule ($\exists\, x : T \bullet x = v \wedge P(x) \ \equiv\ P(v)$) to the quantified variables as' and out we get

$$\forall\, as : AS;\ gs' : GS;\ ls' : LS;\ in : In \bullet$$
$$(gs.c \bmod 2 = 0 \Rightarrow gs.x1 = as.x1 \wedge gs.x2 = as.x2) \wedge$$
$$IN(d1, d2) = IN(in) \wedge ls.pc = 5 \wedge ls'.pc = 6 \wedge$$
$$gs'.c = gs.c + 1 \wedge gs'.x1 = gs.x1 \wedge gs'.x2 = gs.d2$$
$$gs'.lock = gs.lock \Rightarrow$$
$$gs'.c \bmod 2 = 0 \wedge$$
$$(gs'.c \bmod 2 = 0 \Rightarrow gs'.x1 = d1 \wedge gs'.x2 = d2)$$

Applying the one-point rule ($\forall\, x : T \bullet x = v \wedge P(x) \Rightarrow Q(x) \ \equiv\ P(v) \Rightarrow Q(v)$) to the quantified variables gs', ls' and in, we get

$$\forall\, as : AS \bullet$$
$$(gs.c \bmod 2 = 0 \Rightarrow gs.x1 = as.x1 \wedge gs.x2 = as.x2) \wedge$$
$$ls.pc = 5 \Rightarrow$$
$$gs.c \bmod 2 \neq 0 \wedge$$
$$(gs.c \bmod 2 \neq 0 \Rightarrow gs.x1 = d1 \wedge gs.x2 = d2)$$

which (using $A \wedge (A \Rightarrow B) \equiv A \wedge B$) simplifies to

$$\begin{array}{l}\forall\, as : AS \bullet \\ \quad (gs.c \bmod 2 = 0 \Rightarrow gs.x1 = as.x1 \wedge gs.x2 = as.x2) \wedge \\ \quad ls.pc = 5 \Rightarrow \\ \qquad gs.c \bmod 2 \neq 0 \wedge gs.x1 = d1 \wedge gs.x2 = d2\end{array}$$

which (since there is an $as : AS$ which satisfies the first line of the antecedent) simplifies to

$$ls.pc = 5 \Rightarrow gs.c \bmod 2 \neq 0 \wedge gs.x1 = d1 \wedge gs.x2 = d2\,.$$

Note that $d1$ and $d2$ are not free variables. They can be expressed in terms of $status(ls, gs)$ which equals $IN(d1, d2)$.

Continuing in this manner for the remainder of the *write* operation, the required invariant to prove linearizability is

$$\begin{array}{l}(ls.pc = 6 \Rightarrow gs.c \bmod 2 = 0) \wedge \\ (ls.pc = 5 \Rightarrow gs.c \bmod 2 \neq 0 \wedge gs.x1 = d1 \wedge gs.x2 = d2) \wedge \\ (ls.pc = 4 \Rightarrow gs.c \bmod 2 \neq 0 \wedge gs.x1 = d1) \wedge \\ (ls.pc = 3 \Rightarrow gs.c \bmod 2 \neq 0) \wedge \\ (ls.pc = 2 \Rightarrow gs.c \bmod 2 = 0) \wedge \\ (ls.pc = 1 \Rightarrow (\neg\, gs.lock \Rightarrow gs.c \bmod 2 = 0))\,.\end{array}$$

The final conjunct requires the starting invariant $I_0(gs, ls)$ to be true when $ls.pc = 1$. Since this is true when $ls.pc = 0$ it is guaranteed to be true at $ls.pc = 1$, and hence all the required conjuncts will be true during the operation's execution. In general, the starting invariant need only imply the condition that needs to be true at the first line of an operation. Using this invariant one can prove the conditions necessary for the simulation of the *write* operation.

For the *read* operation of *seqlock*, the linearization point is line 11 when $c = c0$. Following the approach above, we obtain the following conjuncts for the invariant:

$$\begin{array}{l}(ls.pc = 12 \Rightarrow (\neg\, gs.lock \Rightarrow gs.c \bmod 2 = 0)) \wedge \\ (ls.pc = 11 \wedge c = c0 \Rightarrow \\ \quad gs.x1 = ls.d1 \wedge gs.x2 = ls.d2 \wedge \\ \quad (\neg\, gs.lock \Rightarrow gs.c \bmod 2 = 0)) \wedge \\ (ls.pc = 11 \wedge c \neq c0 \Rightarrow (\neg\, gs.lock \Rightarrow gs.c \bmod 2 = 0)) \wedge \\ (ls.pc = 10 \Rightarrow \\ \quad (c = c0 \Rightarrow gs.x1 = ls.d1) \wedge \\ \quad (\neg\, gs.lock \Rightarrow gs.c \bmod 2 = 0)) \wedge \\ ls.pc \in \{7, 8, 9\} \Rightarrow (\neg\, gs.lock \Rightarrow gs.c \bmod 2 = 0))\,.\end{array}$$

In this case, there are two conjuncts for $ls.pc = 11$ reflecting its different roles as a linearization step when $c = c0$, and an internal step otherwise.

5. CONCLUSION

In this paper we have presented an approach for systematically generating the invariants required for an existing proof method for linearizability of concurrent objects. The proof method itself is systematic and fully tool-supported, but requires the invariants to be provided by the verifier. Our approach supports the verifier in this task avoiding unnecessary iteration of proofs steps which may otherwise result.

Our future work will look at extending the approach to objects where the linearization point of an operation can not be determined statically, but relies on the execution of other processes [9, 8], and to objects running on weak memory models such as TSO [12]. This work will be based on the modified proof rules for handling such concurrent objects in [5, 6].

Acknowledgements

This work was supported by Australian Research Council (ARC) Discovery Grant DP160102457.

6. REFERENCES

[1] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In W. Damm and H. Hermanns, editors, *CAV 2007*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.

[2] C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In H. Nielson and G. Filé, editors, *SAS 2007*, volume 4634 of *LNCS*, pages 233–238. Springer, 2007.

[3] J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In J. Davies and J. Gibbons, editors, *IFM 2007*, volume 4591 of *LNCS*, pages 195–214. Springer, 2007.

[4] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.

[5] J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisabilty with potential linearisation points. In M. Butler and W. Schulte, editors, *FM 2011*, volume 6664 of *LNCS*, pages 323–337. Springer, 2011.

[6] J. Derrick, G. Smith, and B. Dongol. Verifying linearizability on TSO architectures. In E. Albert and E. Sekerinski, editors, *iFM 2014*, volume 8739 of *LNCS*. Springer, 2014.

[7] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In D. de Frutos-Escrig and M. Nunez, editors, *FORTE 2004*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.

[8] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04*, pages 206–215. ACM Press, 2004.

[9] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[10] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In *Automated Deduction*, pages 13–39. Kluwer, 1998.

[11] G. Schellhorn, H. Wehrheim, and J. Derrick. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. on Computational Logic*, 15(4):31:1–31:37, 2014.

[12] P. Sewell, S. Sarkar, S. Owens, F. Nardelli, and M. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[13] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In J. Torrellas and S. Chatterjee, editors, *PPoPP '06*, pages 129–136. ACM, 2006.