Simpler Backward Simulation Proofs

Simon Doherty Lindsay Groves

School of Engineering and Computer Science
Victoria University of Wellington
New Zealand
{Simon.Doherty,lindsay}@ecs.vuw.ac.nz

Abstract

Backward simulation relations provide a technique for verifying computer systems modelled as labelled transition systems. Recent experience suggests that backward simulation relations are useful in the verification of some highly concurrent systems. Proofs by backward simulation are complicated by the need to show that the simulation relation is total over all reachable states of the system being verified. Some reachable states exhibit complex dependencies between components of the state. We present a technique that reduces proving totality on all reachable states, to proving totality on a subset of the reachable states that are very simple. The technique exploits a very weak property of concurrent systems that we call completability: a system is completable if every operation can finish, but not every operation is required to finish in every execution.

1 Introduction

Transition systems of various sorts are often used to specify the behaviour of complex systems. Using the same formalism to model the implementations of such systems leads to the use of trace inclusion as a natural correctness condition, where traces capture the interesting parts of executions, and are typically obtained by projecting out some of the actions contained in the execution or some part of the states the system passes through.

The most common way to prove trace inclusion is to show that for any execution of the implementation we can construct an execution of the specification which has the same trace and preserves a simulation relation between concrete and abstract states. It is well known that this technique, known as forward simulation, is not sufficient to verify all possible cases of trace inclusion, and that sometimes a complementary technique, known as backward simulation, is required [6, 10].

Various cases requiring backward simulation are often regarded as a theoretical oddity, and some verification methodologies ignore them entirely. Our experience, however, shows that backward simulations are often required in verifying highly concurrent algorithms [4, 2, 5], where because of subtle interactions between processes, the outcome of an attempted operation may be determined later in the implementation than it is in the specification.

The proof obligations for backward simulation are quite similar to those for forward simulation, with suitable adaptations for the change in direction: we need to show that the simulation relation relates initial states of the implementation to initial states of the specification, and that the simulation relation is preserved by appropriately related pairs of actions or action sequences. For backward simulation, however, it is also necessary to prove that the simulation relation is total, in order to ensure that construction of the abstract execution can begin. In some backward simulations, this condition can be quite difficult to prove, because of the need to deal with states where an arbitrary collection of processes may all be at varying stages in the execution of their operations. This problem becomes particularly acute in cases where there are nontrivial dependencies between the local states of each process. One frequently encounters such dependencies when reasoning about highly concurrent data structures.

In this paper, we give an alternative condition, which we call quasi-totality, that provides a significantly easier way to prove the totality condition in important cases. Briefly, a labelled transition system A is extendible to a given set of states T if every finite execution of A can be extended to one which ends in a state in T. If we can choose T so that the simulation relation is total on T, then we say that the simulation relation is quasi-total. In verifying highly concurrent implementations of data types, we choose T to be the set of quiescent states (i.e., those where no operation has begun, but not finished), for which it is easy to show quasi-totality.

Our work is focussed on the verification of non-blocking concurrent objects. That is, objects for which no operation can prevent any other operation from finishing, even when operations are allowed to fail part-way through. For such objects, extendibility to quiescent states is a necessary condition of correctness, and is typically straightforward to verify. For this reason, we believe that quasi-totality constitutes a useful simplification for an important class of verifications using backward simulation.

The rest of the paper is organised as follows. We begin, in Section 2, by introducing formal definitions of labelled transition systems, traces, etc., then in Section 3, we present the proof obligations for backward simulation. In Section 4, we introduce an extends-to relation on sets of states, which we use to formalise the idea that any finite execution can be extended to one that ends in a state with a given property. In Section 5, we define our notion of quasi-totality. In Section 6, we illustrate the use of this technique, by outlining a verification of Bloom’s construction of a “many reader, two writer” register from a pair of “many reader, one writer” registers. We conclude in Section 7.
2 Preliminaries

2.1 Sequences

We regard a sequence as a function over some prefix of \( \mathbb{N} \). We write \( () \) for the empty sequence, \( (x_0, \ldots, x_n) \) for the sequence containing \( x_0, \ldots, x_n \) in the order presented, and \( \alpha \beta \) for the concatenation of sequences \( \alpha \) and \( \beta \). If \( S \) is a set then \( S^n \) is the set of finite sequences with elements from \( S \).

2.2 Labelled transition systems

A labelled transition system (LTS) is a tuple \( L = (S, S_0, E, \rightarrow) \), where \( S \) is a set of states, \( S_0 \subseteq S \) is a set of initial states, \( A \) is a set of actions, \( E \subseteq A \) is a set of external actions, and \( \rightarrow \subseteq S \times A \times S \) is the transition relation. We write \( s, a \rightarrow_L s' \) for \( (s, a, s') \in \rightarrow \).

Actions that are not external (i.e. \( A \setminus E \)) are called internal actions. For an LTS \( L = (S, S_0, A, E, \rightarrow) \), let \( \text{states}(L) = S \), \( \text{init}(L) = S_0 \), \( \text{acts}(L) = A \), \( \text{external}(L) = E \), and \( \text{internal}(L) = A \setminus E \).

An execution fragment of an LTS \( L \) is an alternating sequence of states and actions \( e = (s_0, a_0, s_1, \cdots) \), ending in a state if \( e \) is finite, such that \( s_i, a_i \rightarrow_L s_{i+1} \) for \( s_i, s_{i+1} \in \text{states}(L) \) and \( a_i \in \text{acts}(L) \). An execution of \( L \) is an execution fragment \( (s_0, a_0, s_1, \cdots) \), such that \( s_0 \in \text{init}(L) \). If \( e \) is an execution fragment of \( L \), then \( \text{trace}(e) \) is the sequence of external actions appearing in \( e \). A trace of \( L \) is the trace of some execution of \( L \). We write \( s, a \rightarrow_L s' \) to mean that there is some \( e \), an execution fragment of \( L \), such that \( e = \text{trace}(e) \), the first state of \( e \) is \( s \), and the last state of \( e \) is \( s' \). The set of reachable states of \( L \), denoted \( \text{reach}(L) \), is the set of states appearing in an execution of \( L \). That is, \( \text{reach}(L) \) is the set of states \( s \in \text{states}(L) \) such that there is some execution of \( L \) whose last state is \( s \). The behaviour of an LTS is defined in terms of its traces. For two LTSs, \( A \) and \( C \), we say that \( C \) is traced included in \( A \) iff every trace of \( C \) is a trace of \( A \).

We describe our LTSs using state variables, which we access using dot notation, so that if \( x \) is a state variable, then \( x.s \) denotes the state like \( s \), but with the value of each variable \( x_i \) set to \( v_i \).

2.3 Objects and Operations

We are particularly interested in working with LTSs that represent objects that expose some set of operations to users. The traces of such objects are sequences of invocations and responses, where each invocation or response is indexed by some port. Each invocation/response pair represents an operation executed at the given port, in the interval between the invocation and the response. The ports are often used to represent processes or threads that use the object. Ports may also be used to represent roles or privileges with respect to the object, which may be passed dynamically between processes or threads. (The example in Section 6 is of this kind.)

We say that a response \( r \) matches an invocation \( i \) in a trace if \( r \) and \( i \) are at the same port \( p \), and there is no \( p \)-indexed invocation or response in \( i \) between \( i \) and \( r \). We model objects in such a way that every response in every trace matches an invocation, and no invocations never occur at any port without an intervening response. However, in order to model operations that have begun but not completed, we admit traces in which there are unmatched invocations. We call such invocations pending operations.

We model objects in such a way that any two executions ending in the same state have the same set of pending operations.\(^1\) Thus, we say that an operation is pending in a state if the operation is pending in a trace of the execution leading to the state. We say that a state is quiescent if there are no pending operations.

3 Backward Simulation

Our definition of backward simulation is adapted from \([10]\). Let \( C \) and \( A \) be two LTSs such that \( \text{external}(C) \subseteq \text{external}(A) \). A relation \( R \subseteq \text{states}(C) \times \text{states}(A) \) is a backward simulation from \( C \) to \( A \) if the following conditions hold:

- Every state of \( A \) related by \( R \) to an initial state of \( C \) is an initial state of \( A \). Formally,

\[
\forall c \in \text{init}(C), \quad a \in \text{states}(A) \quad \implies \quad R(c, a) \quad \implies \quad a \in \text{init}(A) \quad (1)
\]

- \( R \) is preserved by external actions:

\[
\forall c, c' \in \text{reach}(C), \quad a \in \text{states}(A) \quad \implies \quad R(c, a) \quad \implies \quad R(c', a) \quad (2)
\]

- \( R \) is preserved by internal actions:

\[
\forall c, c' \in \text{reach}(C), \quad a \in \text{states}(A) \quad \implies \quad R(c, a) \quad \implies \quad R(c', a) \quad (3)
\]

- \( R \) is total on the reachable states of \( C \):

\[
\forall c \in \text{reach}(C), \quad a \in \text{states}(A) \quad \implies \quad R(c, a) \quad (4)
\]

Conditions (1-3) allow us to build an abstract execution \( \alpha \) with the same trace as a concrete execution \( \gamma \), so long as if \( \gamma \) ends in state \( c \), then there is some state \( a \) such that \( R(c, a) \). The proof, which we omit, proceeds by induction on the length of the concrete execution.

The existence of the required abstract state for all concrete executions is guaranteed by the totality condition, condition (4). Thus, conditions (1-4) together guarantee trace inclusion from \( C \) to \( A \).

Note that the totality condition requires that all reachable states of \( C \) have related abstract states. As we shall see in Section 6, it can be inconvenient or difficult to construct for each reachable state a related abstract state. We use the extendibility property to relax the required totality condition. In particular, we show how we can choose a simple subset of the reachable states of \( C \) for which we can easily show the existence of related abstract states.

\(^1\)In our models, each port is associated with a program-counter value in each state, which describes whether there is a pending operation at that port, and if there is, what stage in its execution that operation has reached. Collectively, the program-counter values of each state determine the set of pending operations appearing in executions that end in that state.
4 Extendibility and Completability

Our technique is based on a safety property, which we call \textit{extends-to}. We say that for LTS \( L \), the set of states \( S \) extends-to the set of states \( T \), denoted \( S \rightarrow_{L} T \), if for all \( s \in S \cap \text{reach}(L) \), there exists some execution fragment \( (s_0, a_0, \ldots, s_n) \) of \( L \) such that \( s_0 = s \) and \( s_n \in T \). If \( L \) satisfies \( \text{states}(L) \rightarrow_{L} T \) for some set \( T \), then we say \( L \) is \textit{extendible} to \( T \).

Note that the property \( S \rightarrow_{L} T \) makes no guarantees about whether a state in \( T \) will actually appear in any given execution. For this reason, \( \text{extends-to} \) is a weaker property than the related temporal notions of \textit{leads-to} (often denoted \( \rightarrow \)) and \textit{eventually} (often denoted \( \diamond \)).

\textit{Leads-to} and \textit{eventually} are progress properties, and proofs of these properties depend on the use of fairness assumptions (except in trivial cases). \text{extends-to} is a safety property, and no such fairness assumptions are required in its proof.

We are particularly interested in \textit{extends-to} properties of the form \( \text{states}(L) \rightarrow_{L} T \), where \( T \) is the set of quiescent states. If \( L \) satisfies this property, then we say that \( L \) is \textit{completable}. An object may be completable even if individual operations are not themselves guaranteed to complete. Completability is strictly weaker than any nonblocking progress property. However, completability is not unique to objects with nonblocking progress properties. An object that uses blocking or mutual exclusion can still have the completability property, so long as deadlock is not possible. Thus, completability is a very weak correctness property, amounting to the assertion that all operations are capable of finishing. For this reason, we feel that establishing that an object is completable should be considered a natural component of a verification.

For some objects, completability obviously holds. In particular, objects in which no operation involves loops or recursion are completable, provided there is no possibility of deadlock. The example presented in Section 6 is of this kind. However, in general it is useful to have some rules for proving completability and other extendibility properties.

Extendibility is closed under the obvious reflexive and transitive closure conditions. Thus, for all \( S \subseteq \text{states}(L) \),

\[ S \rightarrow_{L} S \]  

and for all \( S, T, U \subseteq \text{states}(L) \),

\[ S \rightarrow_{L} T \land T \rightarrow_{L} U \implies S \rightarrow_{L} U \]  

Reflexivity and transitivity only enable us to prove extends-to properties that can be realised within a fixed number of steps. We use well-founded partial orders (wpos)\(^3\) to prove \text{extends-to} results that require a finite but indefinite number of steps to realise.

Let \( (O, \preceq) \) be a well founded partial order, let \( f : \text{states}(L) \rightarrow O \), and let

\[ \text{min}(O, \preceq) = \{ o \in O \mid \forall o' \in O \bullet o' \prec o \} \]

\[ \text{at}_f(o) = \{ s \in \text{states}(L) \mid f(s) = o \} \]

\[ \text{below}_f(o) = \{ s \in \text{states}(L) \mid f(s) < o \} \]

\(^2\)Leads-to and \textit{eventually} are defined over execution fragments, rather than individual states. So strictly speaking, these notions are not directly comparable to \textit{extends-to}. However, it would be straightforward to define \textit{extends-to} over execution fragments by applying our present definition to the first state of such a fragment. We could then make these comparisons precise.

\(^3\)A wpos is a partial order in which there are no infinite descending chains \( x_0 > x_1 > \ldots \).

We can show, using well-founded induction, that

\( (\forall o \in O \bullet o \not\in \text{min}(O, \leq) \implies \text{at}_f(o) \rightarrow_{L} \text{below}_f(o)) \implies \forall o \in O, \exists o' \in \text{min}(O, \leq) \bullet \text{at}_f(o) \rightarrow_{L} \text{at}_f(o') \)  

\( (7) \)

\( (7) \) allows us to prove

\( (\forall o \in O \bullet o \not\in \text{min}(O, \leq) \implies \text{at}_f(o) \rightarrow_{L} \text{below}_f(o)) \implies \text{states}(L) \rightarrow_{L} \{ s \mid \exists o' \in \text{min}(O, \leq) \bullet s \in \text{at}_f(o') \} \)  

\( (8) \)

which is useful in proving completability and other statements about the possibility of termination.

4.1 Extendibility and Temporal Logic

Extendibility can be thought of as a \textit{temporal property}, which describes what is possible in the future of an execution. Thus, extendibility invites comparison with properties definable in temporal logic. There is not sufficient space to give a comprehensive or formal treatment. However, we briefly describe how extendibility relates to \textit{linear-time temporal logic} (LTL) and computation-tree logic (CTL). We assume that the reader has some familiarity with the syntax and semantics of these logics.

LTL formulae describe properties of executions fragments of transition systems. Assertions in LTL about what can happen in the future of an execution are about what \textit{eventually} does happen. Contrast this with extendibility, which can be thought of as an assertion about what \textit{might} happen, but is not guaranteed to do so. Put differently, LTL assertions do not distinguish executions in which something did not happen but could have happened, from executions in which something did not happen but couldn’t have happened. For essentially this reason, extendibility is not definable in LTL.

CTL formulae describe properties of \textit{states}, and the execution fragments of the given transition system that begin in each state. Because of this, CTL is able to express extendibility. In CTL, the formula \( \text{AG} P \) asserts that the property \( P \) is true in every state of every execution starting with the current state. The formula \( \text{EF} P \) asserts that there is some execution starting with the current state, such that \( P \) is eventually true in that execution. Let \( \text{Init}_L \) be a CTL formula characterising the initial states of \( L \), and let \( P_S \) (resp. \( P_T \)) be a CTL formula characterising the states in the set \( S \) (resp. the set \( T \)). Then \( S \rightarrow_{L} T \) can be expressed in CTL as

\[ \text{Init}_L \implies \text{AG} (P_S \implies (EF P_T)) \]

5 Quasi-totality

Using extendibility, we can relax condition (4) so that we only need to prove totality of \( R \) over some set \( T \subseteq \text{states}(C) \), such that \( \text{states}(C) \rightarrow_{L} T \). The idea is that \( T \) should be chosen to exclude problematic or complicated elements of \text{reach}(C).

Let \( R \subseteq \text{states}(C) \times \text{states}(A) \) and \( T \subseteq \text{states}(C) \), Conditions (1-3) imply trace inclusion when combined with the following quasi-totality condition:

**Definition 1 (Quasi-totality)** \( R \) is quasi-total for \( C \) iff there exists some \( T \subseteq \text{states}(C) \) such that

\[ \text{states}(C) \rightarrow_{C} T \]  

\( (9) \)
Note that a write operation can be called at any port, and such that $R$ is quasi-total for $C$. Then

$$\forall c \in T, \exists a \in states(A) \cdot R(c, a) \quad (10)$$

**Lemma 1** Let $R$ be a relation satisfying (1-3) above, and such that $R$ is quasi-total for $C$. Then

$$\forall c \in reach(C), \exists a \in states(A) \cdot R(c, a) \quad (11)$$

**Proof** Fix a state $c \in reach(C)$. The quasi-totality of $R$ implies that there is some $T \subseteq states(C)$ and $\gamma = \langle s_0, act_0, \ldots, s_n \rangle$ such that $s_0 = c$ and $s_n \in T$, and $states(C) \rightarrow c \cdot T$. We prove by induction on the length of $\gamma$ that there is some $a \in states(A)$ such that $R(c, a)$. This is sufficient to prove the lemma. If $\gamma = \langle c \rangle$, then $c \in T$, so let $a = \langle a \rangle$, where $a$ is an abstract state such that $R(c, a)$, whose existence is guaranteed by (10). Now, let $\gamma = \langle c, act \rangle \circ \gamma'$, and let $c'$ be the first state of $\gamma'$. By hypothesis, there is some state $a'$ such that $R(c', a')$. If $act$ is internal, then by condition (3), there is some state $a$ such that $R(c, a)$. Likewise, if $act$ is external, then by condition (2), there is some such state $a$. This completes the proof.

Lemma 1 allows us to conclude that the existence of a relation $R$ satisfying conditions (1-3) and a set $T$ satisfying (9-10) is enough to show that the traces of $C$ are all traces of $A$.

6 Bloom’s Register

We illustrate the utility of our approach with an example. Bloom [1] describes a construction of a register that allows any number of concurrent read operations, and up to two concurrent write operations. We call such a register a two-writer register. The construction assumes the existence of two registers, each of which allows any number of concurrent read operations, and one write operation at a time. We call such registers single-writer registers. The original proof of this construction [1] is complicated, and is not based on any clear and well-understood verification methodology. The construction was first used as an example of an “assertional” verification technique by Hesselink [9], who employed a correctness criterion specialised for the verification of registers offering read and write operations. 

6.1 The Specification

We specify the set of correct behaviours of registers providing read and write operations (henceforth read/write registers) using the LTS $A$, whose transition relation is presented in Figure 1. The register ranges over values from some set $V$. The external actions of $A$ represent invocations and responses of read and write operations, each indexed by a port from the set $P = N$. That is,

$$\text{external}(A) = P \times \{\text{writeInv}(v) \mid v \in V\} \cup \{\text{readResp}(v) \mid v \in V\} \cup \{\text{writeResp}(a, \text{reg})\} \quad (12)$$

Note that a write operation can be called at any port, so that $A$ is more general than the two-writer register. $A$ has two sets of internal actions, each indexed by a port number: $\text{doRead}_p$ and $\text{doWrite}_p(v)$, for each $v \in V$ and $p \in P$. Each “do” action indexed by port $p$ represents the instant when the operation associated with $p$ “appears to take effect”, from the perspective of an external observer of the system. This point is known as the linearisation point [8] of the operation, and the traces of $A$ are linearisable. Linearisability is the standard notion of correctness for read/write registers, and shared-memory concurrent objects in general. Informally, a trace is linearisable if each operation “appears to take effect” at some point between the operation’s invocation and response. More formally, an execution is linearisable with respect to a sequential specification (such as read/write registers), if there is some sequential order on the operations that is allowed by the sequential specification, and such that if an operation $o$ completes in the concurrent execution before another operation $o'$ begins, then $o$ occurs before $o'$ in the sequential order. It can be shown that our LTS $A$ captures precisely the set of linearisable traces for read/write registers.$^5$

Each port number has an associated program counter variable $pc_p$ which is used in such a way that each operation at port $p$ first takes an invocation action, then a “do” action, then a response action. Between operations we set $pc_p$ to a value indicating that the port $p$ is “idle”. Furthermore, each doWrite action sets a variable $\text{reg}$ to the value being written, and each doRead action reads the current value of $\text{reg}$, and arranges for that value to be “returned” at the next response action. A state $a$ is in start($A$) iff for all $p \in P$, $pc_p = \text{idle}$. (So we do not specify an initial value for the register.)

6.2 The Implementation

Figure 2 presents the transition relation of the LTS $C$ modelling Bloom’s construction. The external actions of $C$ are the external actions of $A$, except that the indexes of the writeInv and writeResp actions are constrained to be in the set $\{0, 1\}$. Bloom’s construction assumes two single-writer registers, denoted $\text{Reg}[0]$ and $\text{Reg}[1]$, each of which allows any number of concurrent read operations, and one write operation.

$^5$In fact, $A$ is a canonical automaton [11] for read/write registers.
c, writeInv_p(v) \rightarrow c' \iff \\
c.pcp_p = idle \wedge \\
c' = c \oplus \{pc_p \mapsto wr0(v)\} \\
c, readInv_p \rightarrow c' \iff \\
c.pcp_p = idle \wedge \\
c' = c \oplus \{pc_p \mapsto rd0\} \\
c, writeResp_p \rightarrow c' \iff \\
c.pcp_p = writeResp \wedge \\
c' = c \oplus \{pc_p \mapsto idle\} \\
c, readResp_p(v) \rightarrow c' \iff \\
c.pcp_p = readResp(v) \wedge \\
c' = c \oplus \{pc_p \mapsto idle\} \\
c, wr0_p(v) \rightarrow c' \iff \\
c.pcp_p = wr0(v) \wedge \\
c' = c \oplus \{pc_p \mapsto wr1(\pi_1(c.\text{Reg}[p \oplus 1]),v)\} \\
c, wr1_p(b,v) \rightarrow c' \iff \\
c.pcp_p = wr1(b,v) \wedge \\
c' = c \oplus \{pc_p \mapsto writeResp, \\
c.\text{Reg}[p] \mapsto (b \oplus p, v)\} \\
c, rd0_p \rightarrow c' \iff \\
c.pcp_p = rd0 \wedge \\
c' = c \oplus \{pc_p \mapsto rd1(\pi_1(c.\text{Reg}[0]))\} \\
c, rd1_p(b_0) \rightarrow c' \iff \\
c.pcp_p = rd1(b_0) \wedge \\
c' = c \oplus \{pc_p \mapsto rd2(b_0, \pi_1(c.\text{Reg}[1]))\} \\
c, rd2_p(b_0, b_1) \rightarrow c' \iff \\
c.pcp_p = rd2(b_0, b_1) \wedge \\
c' = c \oplus \{pc_p \mapsto readResp(\pi_2(c.\text{Reg}[b_0 \oplus b_1]))\}

Figure 2: The transition relation of the LTS C, modelling Bloom’s Construction. In this model, the port numbers of the actions writeInv, writeResp, wr0 and wr1 must be less than 2.

tag from the “other” register, and then modifies its “own” register. Observe that in the special case where no operations overlap, after each write operation indexed by \(w \in \text{Writers}\), we have \(w = cur(c)\). Thus, in this special case, each write operation successfully modifies the value of the two-writer register.

Each read operation is indexed by a port number in the set \(\{a \mid 2 \leq n\}\). Each read operation proceeds by reads the tag of \(\text{Reg}[0]\), then the tag of \(\text{Reg}[1]\), and returning the value in the register indexed by the modulo-2 sum of these values. Thus, each read operation that does not overlap with a write operation returns the current value of the two-writer register.

When operations overlap, the situation is more complicated. We follow Bloom [1] in distinguishing two kinds of writes. We say that a write operation indexed by \(w \in \{0, 1\}\) is potent if, immediately after executing \(wr1\), we have \(w = cur(c)\), and impotent otherwise. The linearisation point of each potent write operation is its \(wr1\) action.

Each read operation that returns a value written by a potent write is linearised either at the read operation’s \(rd0\) action, or in the case where the read operation overlaps the potent write, just after the linearisation point of the potent write.

It can be shown that each impotent write overlaps a potent write at the other port. We choose as the linearisation point of each impotent write a point just before the \(wr1\) action of the first overlapping potent write.

It can be shown that each read operation that returns a value written by an impotent write overlaps both the impotent write and the potent write that provides the impotent write’s linearisation point. The linearisation point of each such read is a point between the linearisation points of the impotent write and the potent writes.

### 6.3 Constructing a Backward Simulation

Condition (3) implies that if there is a backward simulation between \(C\) and \(A\), then for each operation there is some internal action of \(C\) which corresponds to a sequence of internal actions of \(A\) that contains the “do” action of the operation. The execution of this internal action is a linearisation point for the operation. Thus, the linearisation points of an object’s operations induce a scheme for choosing internal action sequences to satisfy condition (3). In turn, this scheme induces an outline of the relation itself. Figure 3 illustrates how we choose abstract actions in an execution containing both potent and impotent writes. Let \(wr_p, wr_1 \in \text{Writers}\) and \(rd_p, rd_1 \in \text{Readers}\). Note that the \(wr_1\) action of \(wr_p\) is between the \(wr0\) and \(wr1\) actions of \(wr_1\). This makes \(wr_p\) a potent write, and \(wr_1\) impotent. In this scenario, \(rd_p\) reads \(wr_p\’s\ write, and \(rd_1\) reads that of \(wr_1\). Because of this, we associate \(wr_p\’s\ internal action \(wr1\) with the sequence \(\alpha\) containing “do” actions for all four ports.

We now present part of our backward simulation relation for the verification of Bloom’s register construction. We do not present the entire relation, nor argue that it is in fact a backward simulation. We limit ourselves to describing some of the properties that it must have. Collectively, these properties constitute a relation over concrete and abstract states that is weaker than the actual backward simulation. As an illustration of the simplifying power of using the quasi-totality property rather than totality, we compare the proofs of the quasi-totality and totality properties for this weaker relation. For reasons discussed shortly, the quasi-totality proof is no harder for the full simulation than for the relation described...
our backward simulation must have the property that
\[ a.p \rightarrow \forall p \in \{ \text{doRead}, \text{doWrite} \}, \forall \alpha \in \{ \text{wr0}, \text{wr1} \}, (a.p) \circ \alpha = \text{wr1}(b, v) \wedge \text{cur}(c) \neq p \odot b \odot \pi_1(c.p) \rightarrow a.p \neq \text{writeResp} \] for all related states in the abstract sequence containing the doWrite operation at p’s linearisation point. Furthermore, we must be able to show that when p ∈ Writers takes its wr0 action, p’s program counter value in the abstract state is doWrite(v) for the value v being written by p. Otherwise, we will not be able to prove that condition (19) holds for the poststate of every wr0 action. The poststate of the wr0 action for port p has the property that
\[ \text{cur}(c) = p \odot b \odot \pi_1(c.p) \]

Therefore, our backward simulation relation should imply that, for all related c ∈ states(A) and a ∈ states(A),

\[ a.p \rightarrow \forall p \in \{ \text{doRead}, \text{doWrite} \}, b \in \{ 0, 1 \}, v \in V \bullet \]
\[ a.p \neq \text{writeResp} \]

This is so that we can choose an abstract action sequence containing the doWrite operation at p’s linearisation point.

Similarly, we must associate a readInv or writeInv action with the same abstract action. Therefore, our backward simulation must ensure that for all related states in the abstract sequence containing the doWrite operation at p’s linearisation point. Furthermore, we must be able to show that when p ∈ Writers takes its wr0 action, p’s program counter value in the abstract state is doWrite(v) for the value v being written by p. Otherwise, we will not be able to prove that condition (19) holds for the poststate of every wr0 action. The poststate of the wr0 action for port p has the property that
\[ \text{cur}(c) = p \odot b \odot \pi_1(c.p) \]

Therefore, our backward simulation relation should imply that, for all related c ∈ states(A) and a ∈ states(A),

\[ a.p \rightarrow \forall p \in \{ \text{doRead}, \text{doWrite} \}, b \in \{ 0, 1 \}, v \in V \bullet \]
\[ a.p \neq \text{writeResp} \]

This is so that we can choose an abstract action sequence containing the doWrite operation at p’s linearisation point. Furthermore, we must be able to show that when p ∈ Writers takes its wr0 action, p’s program counter value in the abstract state is doWrite(v) for the value v being written by p. Otherwise, we will not be able to prove that condition (19) holds for the poststate of every wr0 action. The poststate of the wr0 action for port p has the property that
\[ a.p \neq \text{writeResp} \]

Recall that each p ∈ Readers is linearised at some point during the execution of the write operation that wrote the value returned by p, or else p does not overlap that write, and is linearised at its rd0 action. To keep track of the situations under which these linearisations occur, our backward simulation must have the property that for all related c ∈ states(C) and a ∈ states(A),

\[ a.p \rightarrow \forall p \in \{ \text{doRead}, \text{doWrite} \}, b \in \{ 0, 1 \}, v \in V \bullet \]
\[ a.p \neq \text{writeResp} \]

It can be shown that for each port p ∈ Writers, if p reaches a state c where c.pc_wr = wr1(b, v) and
\[ \text{cur}(c) \neq p \odot b \odot \pi_1(c.p) \]
then p’s write is doomed to be impotent. Therefore, our backward simulation must have the property that
\[ a.p \neq \text{writeResp} \]
This introduces a dependency between the local states of the read ports, and the local states of the write ports.

6.4 Totality Proofs

The complete simulation relation for Bloom’s construction requires several other constraints, which are not important for our present purpose. We note only that none of these properties add any additional constraint to any port $p$ such that $c.pcp_p = idle$, nor do they require any strengthening of the relationship between $a.reg$ and $c.Reg$ given in condition (15).

Let $R$ be the relation defined by the conjunction of conditions (15-23).

Lemma 2 $R$ is quasi-total for $C$.

Proof Let

$$ T = \{ c | \forall p \in P \land c.pcp_p = idle \} $$

and fix $c \in T$. Let $a$ be the abstract state such that $a.reg = Value(c)$, and

$$ \forall p \in P \land a.pcp_p = idle $$

Then $R(c,a)$. To see this, all that must be checked are conditions (15) and (16). To see that $states(C) \rightarrow C$ $T$, observe that for each port $p$, if $c.pcp_p \neq idle$, then $p$ may take an action, and furthermore $p$ may take at most three actions before completing its operation. This completes our proof.

Note that embedded in the (very simple) proof of Lemma 2, is a proof that each operation completes in finitely many of its own steps. This property is known as wait-freedom [7], and it is the strongest of the nonblocking progress properties. We reiterate that all nonblocking progress properties are strictly stronger than completable, and therefore we obtain the completable property of the quasi-totality lemma as an immediate consequence of the progress conditions required of the concurrent objects whose verification motivates this work.

It is also important to note that no matter what we add to $R$ to arrive at a simulation relation, so long as we do not further constrain ports in the idle state or modify the relationship between $a.reg$ and $c.Reg$, the difficulty of the quasi-totality proof does not increase. We only need to check the same simple properties. This is in contrast to the direct proof that $R$ is total over all the reachable states of $C$. As we shall see, any extension to $R$ would make this proof longer, and potentially more complicated.

Lemma 3 (Totality of $R$) For all states $c \in reach(C)$, there exists an $a \in states(A)$ such that $R(c,a)$.

Proof Fix $c \in reach(c)$. We must take more care with our choice of abstract state $a$. First, as before, $a$ must satisfy

$$ a.reg = Value(c) $$

For each $p$ such that

$$ c.p cp_p \in \{ idle, writeResp, readInv \} \cup \{ writeInv(v), readResp(v) \mid v \in V \} $$

set $a.pcp_p = c.pcp_p$. These constraints are sufficient to allow us to prove conditions (16-20). The situation is more complicated for ports with other program-counter values. For all $p \in Writers$, if $c.pcp_p = wr0(v)$ or $c.pcp_p = wr1(b, v)$ and

$$ cur(c) = p \odot b \odot \pi_1(c.Reg[p \odot 1]) $$

then set $a.pcp_p = doWrite(v)$. Otherwise if $c.pcp_p = wr1(b, v)$, set $a.pcp_p = writeResp$. This allows us to prove conditions (21-22). To satisfy condition (23), we can get away with setting $a.pcp_p = doRead$ whenever $c.pcp_p = rd2(b0, b1)$. Apart from the tedium arising from the need to treat several different cases, there are two drawbacks about the proof of full totality, relative to the proof of quasi-totality. First, the difference in the effort required to mechanise the proofs of Lemmas 2 and 3 is substantial. Our verification work is directed towards the construction of proofs that can be checked by a mechanical proof assistant, such as PVS [3]. PVS is unable to infer which abstract state $a$ should be constructed for a given concrete state $c$ (and we expect that other proof assistants would have a similar limitation). Therefore, the human operator must manually construct this state. This amounts to enumerating each case given above, but with greater detail and formality. Second, it is not in general possible to avoid dealing with dependencies between ports, as we did in the case where $c.pcp_p = rd2(b0, b1)$. For example, the (far more complicated) backward simulation presented by Doherty and Moir [5] involves such unavoidable dependencies. In that case, it is necessary to tailor the local abstract states of some processes to satisfy requirements of other processes, and showing that these requirements can always be satisfied involves sophisticated reasoning about the invariants of the implementation. In fact, the difficulties associated with mechanising the proof of the totality lemma had substantial impact on the design of the backward simulation relation itself.

7 Concluding Remarks

We have presented a simplification for proofs that use backward simulation. The simplification is technically straightforward, but has significant practical implications for those involved in the verification of highly concurrent data structures. Our previous work [4, 2, 5] has shown that backward simulations arise with unusual frequency in the context of these data structures. The simplification that we have presented is especially useful when the proofs must be mechanised.

Our simplification works by replacing the sometimes onerous totality condition by a weaker quasitotality condition. For the verifications that we are interested in, this replacement is sound given a simple but important correctness criterion, which we have called completable. As we have noted, completable is weaker than other important correctness conditions, including deadlock freedom and all nonblocking progress properties. Thus, our simplification can be applied “for free” after establishing one or more of these stronger properties.

References

Proceedings of the 18th International Conference on Computer Aided Verification (CAV ’06), 2006.


