

Checking Progress with Action Priority: Is it Fair?

Dimitra Giannakopoulou, Jeff Magee, and Jeff Kramer

Department of Computing, Imperial College of Science, Technology
and Medicine, 180 Queen's Gate, London SW7 2BZ, UK
{dgl, jnm, jk}@doc.ic.ac.uk

Abstract. The liveness characteristics of a system are intimately related to the notion of fairness. However, the task of explicitly modelling fairness constraints is complicated in practice. To address this issue, we propose to check LTS (Labelled Transition System) models under a strong fairness assumption, which can be relaxed with the use of action priority. The combination of the two provides a novel and practical way of dealing with fairness. The approach is presented in the context of a class of liveness properties termed *progress*, for which it yields an efficient model-checking algorithm. Progress properties cover a wide range of interesting properties of systems, while presenting a clear intuitive meaning to users.

1 Introduction

Our research objective is the development of practical and effective techniques for modelling and analysing the behaviour of concurrent systems. We aim to support analysis based on the software architecture of a system, and believe that the analysis techniques need to be both accessible to practising software engineers, and supported by powerful automated tools. In particular, our approach is based on the use of Labelled Transition Systems (LTS) to specify behaviour and Compositional Reachability Analysis (CRA) to check composite system models. The architecture description of a system drives CRA in generating the model of the system based on its components [14, 22, 23]. The model thus generated can be checked against the properties required of it.

Previous papers have addressed the problem of verifying safety and liveness properties in the context of CRA [6, 7]. Our work on liveness property checking [6] takes the automata-theoretic approach to verification [30], adopted in a number of existing methods and tools [1, 15, 16]. The approach is based on the use of Linear Temporal Logic (LTL) formulas or Büchi automata to represent liveness properties. The LTS of a program is converted into a Büchi automaton and the LTL formula for some property F is translated into the Büchi automaton for $\neg F$. The automaton corresponding to the intersection of the system and the automaton obtained for $\neg F$ is then constructed. If the resulting automaton is empty then the property F is not violated.

The tractability of the method is significantly affected by the fact that the Büchi automaton B is composed with the system. The size of the system can thereby increase by m times in the worst case, where m is the size of B . Moreover, the size of a Büchi automaton may increase exponentially as a function of the length of the LTL formula that it represents [16]. Although efficient algorithms exist for the automatic translation of LTL formulas into Büchi automata [12], none of these algorithms can guarantee to generate the minimal automaton. In such a setting, fairness is usually represented in terms of constraints introduced in the form of Büchi automata, which are also composed with the system [1, 16]. Besides complicating the task of modelling, this may further increase the size of the system to be analysed.

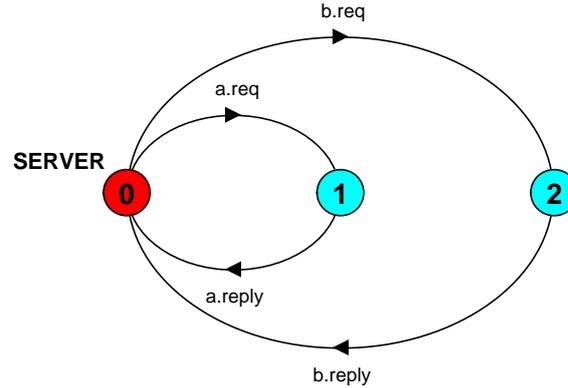
To avoid burdening the users with modelling fairness constraints explicitly, we propose an optional predefined fairness assumption on the executions of an LTS model. Under this assumption, we have found that a specific class of liveness properties, which we have termed progress, can be checked on the unmodified LTS of the system. This is an advantage compared to methods that may increase the state space of the system by the introduction of property and fairness automata. As the fairness assumption may be too restrictive in some circumstances, we introduce an action priority scheme that relaxes it. This combination provides a simple, practical and effective way of dealing with selected types of liveness, and of taking fairness into account when performing liveness property checks. Most importantly, the technique is widely accessible since it requires little or no experience with temporal logic.

Note that the class of liveness properties that can be expressed as progress properties is a subset of those that can be expressed with LTL. Consequently, we do not see progress as supplanting the need for general LTL model checking. We simply propose it as a more accessible alternative to Büchi automata, whenever it covers the particular needs of the system developer. As discussed later in the paper, our experience and that of others [11] indicate that a large number of interesting properties of systems can be expressed and checked in terms of progress properties.

The LTSA tool. The results of our work have been incorporated in an analysis tool – the Labelled Transition System Analyser (LTSA) [22, 23]. The examples used in the paper to illustrate progress checking were developed using the LTSA tool. We will briefly present how models of system behaviour are described for the LTSA. The tool uses a simple process algebra notation, called FSP for Finite State Processes, to define the behaviour of processes. As an aid to understanding, the LTSA supports the facility of drawing the LTS corresponding to an FSP specification.

Fig. 1 gives the FSP specification and corresponding LTS of a server that may be accessed by two clients, A and B . The server may receive requests by either client A or client B (actions $a.req$ and $b.req$, respectively). After receiving a request, the server processes it and produces a corresponding reply (actions $a.reply$ and $b.reply$). The behaviour of $SERVER$ is defined using action prefix (“ $->$ ”), choice (“ $|$ ”) and recursion. In the interests of brevity, we will not formally define their semantics here; the meaning in the example should be clear from the associated LTS diagram.

Structure. The next section describes how progress properties are specified, and how they are checked under the proposed fairness assumption. Section 3 introduces the concept of action priority and its use in progress analysis. Section 4 presents the Readers/Writers example that is used to illustrate and evaluate our approach. Finally, section 5 discusses related work, and section 6 closes the paper with conclusions and plans for future work.



FSP: $SERVER = (a.req \rightarrow a.reply \rightarrow SERVER \mid b.req \rightarrow b.reply \rightarrow SERVER)$.

Fig. 1. FSP specification and LTS for process *SERVER*

2 Progress Properties and the Need for Fairness

The regular occurrence of some actions in a system execution indicates that system behaviour progresses as desired or expected. We would therefore like to be able to check on the model of a system that, in all possible executions of the system, such actions occur regularly. In the context of an infinite execution, regularly means infinitely often. A property that asserts that an action a is expected to occur infinitely often in every infinite execution of the system is expressed in LTL as $\Box\Diamond a$. We call properties of this type *progress*. Often, progress is not determined by a single action but by one of a set of alternatives. For example, a system may be considered to make progress if it outputs one of a set of values. Consequently, we define progress properties in terms of a finite set of actions as follows:

progress $P = \{a_1, a_2, \dots, a_n\}$ defines a progress property P which asserts that in any infinite execution of a target system, at least one of the actions a_1, a_2, \dots, a_n occurs infinitely often.

The LTL formulation of the progress property P is $\Box\Diamond(a_1 \vee a_2 \dots \vee a_n)$. Consider a very simple system S that consists of the server modelled in Fig. 1, and two clients A and B accessing it. The system can be expressed as the parallel composition of the two clients and the server, as illustrated in Fig. 2. Processes assembled with the \parallel parallel

composition operator run concurrently by synchronisation on actions that are common to their alphabets and interleaving of the remaining actions. The LTS for system S is identical to the LTS of Fig. 1.

$$\begin{aligned} A &= (a.req \rightarrow a.reply \rightarrow A). \\ B &= (b.req \rightarrow b.reply \rightarrow B). \\ ||S &= (A || B || SERVER). \end{aligned}$$

Fig. 2. FSP specification of a system with two clients accessing a server

For system S it is likely that a designer would expect both progress properties $SERVE_A$ and $SERVE_B$ to hold, where:

$$\begin{aligned} \text{progress } SERVE_A &= \{a.reply\} \\ \text{progress } SERVE_B &= \{b.reply\} \end{aligned}$$

The reason is that an execution where the requests of some client are ignored indefinitely is clearly undesirable.

These properties do not hold for S (its LTS is identical to that of Fig.1). For example, $SERVE_B$ is violated because S can generate an infinite execution that only listens to the requests of client A by always choosing the transition leading to state (1) when at state (0). This violation corresponds to a scheduler that is consistently biased against a specific enabled transition when given a choice. However, any reasonable scheduler should implement some notion of fairness when choosing between sets of possible transitions. As it is not possible to express fairness explicitly in the standard LTS model, we make the following fairness assumption in order to check progress:

Fair Choice: If a choice over a set of transitions is executed infinitely often, then every transition in the set is executed infinitely often.

As discussed in section 5, fair choice corresponds to a strong fairness assumption on the system transitions. Under fair choice, progress properties $SERVE_A$ and $SERVE_B$ hold for system S . Consider now the case where in system S , client B is substituted by client B_FAULTY that may crash as modelled by the following FSP expression:

$$B_FAULTY = (b.req \rightarrow b.reply \rightarrow B_FAULTY \mid b.crash \rightarrow STOP).$$

For simplicity, we assume that B_FAULTY does not crash while waiting for a reply. The LTS of system S in this case is illustrated in Fig. 3. We can see that in this system, progress property $SERVE_B$ is no longer satisfied: action $b.reply$ can only occur finitely many times in any *fair* infinite execution that reaches states (1) or (2) at some point. The set of states $\{1,2\}$ is called a terminal set of states, because each state is mutually reachable, but no state outside the terminal set can be reached from any of those states. We will prove later that in finite state systems, any fair infinite execution reaches a terminal set of states. As a result, the only actions that are repeated infinitely often in such executions are actions that label transitions between states of the terminal set.

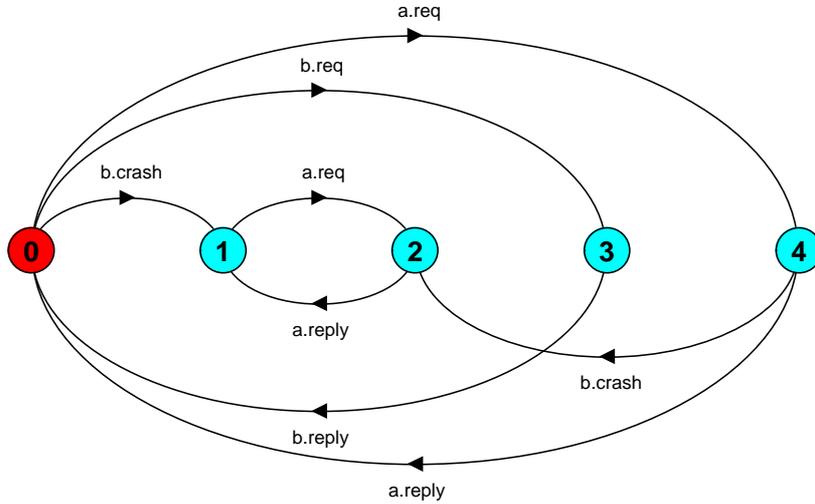


Fig. 3. System consisting of a server and two clients, one of which may crash

The LTSA tool reports the violation as follows:

```

Progress violation: SERVE_B
Trace to terminal set of states:
  a.req
  b.crash
Actions in terminal set:
  {a.req, a.reply}
  
```

This violation does not correspond to a real problem with the system. It is obvious that reply actions cannot occur infinitely often if, after some point, requests are no longer being issued. So the desired property is in fact that, if requests from B occur regularly, then replies to B must also occur regularly, i.e. $\Box\Diamond b.req \Rightarrow \Box\Diamond b.reply$. We call this form of progress property *conditional progress*, which we define as follows:

progress $P = \mathbf{if} \{a_1, a_2..a_n\} \mathbf{then} \{b_1, b_2..b_n\}$
 defines a progress property P which asserts that in any infinite execution of a target system, if any of the actions $a_1, a_2..a_n$ occurs infinitely often then at least one of the actions $b_1, b_2..b_n$ also occurs infinitely often.

Progress property $SERVE_B$ can therefore be restated as follows:

```

progress SERVE_B = if {b.req} then {b.reply}
  
```

This property is satisfied by system S , since after B_FAULTY crashes, it stops making requests to the server. The property therefore makes sure that, when B_FAULTY is

alive, its requests are never consistently ignored, which is what the user wishes to check ¹.

In the following, we formally describe and prove the checking mechanism for progress properties for a system executing under fair choice.

Labelled Transition Systems:

Let *States* be the universal set of states, *Act* be the universal set of observable action labels, and $Act_\tau = Act \cup \{\tau\}$, where τ is used to denote an action that is internal to a subsystem, and therefore unobservable by its environment. An LTS of a process *P* is a quadruple $\langle S, A, \Delta, q \rangle$ where:

- $S \subseteq States$ is a finite set of states,
- $A = \alpha P \cup \{\tau\}$, where $\alpha P \subseteq Act$ is the communicating *alphabet* of *P*,
- $\Delta \subseteq S \times A \times S$, is a transition relation that maps a state and an action onto another state,
- $q \in S$ indicates the initial state of *P*.

For an LTS $P = \langle S, A, \Delta, q \rangle$, we say that action $a \in A$ is *enabled* at a state $s \in S$, iff $\exists s' \in S$ such that $(s, a, s') \in \Delta$. Similarly, we say that a transition $(s, a, s') \in \Delta$ is enabled at a state $t \in S$ iff $t = s$.

We call an *execution* of *P* an infinite sequence $q_0 a_0 q_1 a_1 \dots$ of states q_i and actions a_i , such that $q_0 = q$ and $\forall i \geq 0, (q_i, a_i, q_{i+1}) \in \Delta$. A *trace* of *P* is a sequence of observable actions that *P* can perform starting from its initial state [17].

A state r is *reachable* from a state s in an LTS $P = \langle S, A, \Delta, q \rangle$, iff $((r = s) \text{ or } (\exists a \in A \text{ and } t \in S, \text{ such that } (s, a, t) \in \Delta \text{ and } r \text{ is reachable from } t))$. For a state $s \in S$, $Reachable(s, P)$ denotes the set of states that are reachable from s in *P*, i.e. $Reachable(s, P) = \{r \in S \mid r \text{ is reachable from } s \text{ in } P\}$. An LTS of $P = \langle S, A, \Delta, q \rangle$ transits into another LTS of $P' = \langle S, A, \Delta, q' \rangle$ with an action $a \in A$ iff $(q, a, q') \in \Delta$. That is:

- $\langle S, A, \Delta, q \rangle \xrightarrow{a} \langle S, A, \Delta, q' \rangle$ iff $(q, a, q') \in \Delta$.

Definition – A *terminal set of states* $C \subseteq S$ in an LTS $P = \langle S, A, \Delta, q \rangle$ is a strongly connected component with no outgoing transitions i.e.

- $\forall s \in C, C \subseteq Reachable(s, P)$, and
- $\forall s \in C, Reachable(s, P) \subseteq C$. ■

It follows directly from the above definition that C is a terminal set of states in an LTS *P* iff $\forall s \in C, Reachable(s, P) = C$.

Terminal Set Theorem – Let $P = \langle S, A, \Delta, q \rangle$ be a finite-state process that executes under “fair choice”. If w is a legal infinite execution of *P*, then the set of states that appear infinitely often in w forms a terminal set of states in *P*.

¹ If in addition we wanted to check that a reply is received for *each* request, we would combine the progress property with a safety property [7], which would ensure that a reply must occur in any interval defined by two requests.

Proof: Let $S_1 \subseteq S$ be the set of states that occur infinitely often in w . Since P consists of a finite number of states, then S_1 is not empty. With fair choice, the fact that states in S_1 are repeated infinitely often in w implies that all transitions that are enabled at these states also occur infinitely often in w . This means that all states that are reachable from states of S_1 in P occur infinitely often in w . We conclude that $\forall s \in S_1, \text{Reachable}(s, P) \subseteq S_1$. It is also straightforward that since all states in S_1 are repeated infinitely often in w , then every state in S_1 is reachable from any other state in S_1 , and therefore $\forall s \in S_1, S_1 \subseteq \text{Reachable}(s, P)$. We conclude that $\forall s \in S_1, \text{Reachable}(s, P) = S_1$ and therefore S_1 is a terminal set of states. ■

From the Terminal Set Theorem we conclude that a fair infinite execution w is obtained by repeating infinitely often states in a terminal set of states. As a result, the actions that occur infinitely often in w are exactly those actions that are enabled at states in the terminal set. Therefore, a property “progress $P = \{a_1, a_2 \dots a_n\}$ ” is satisfied iff for each terminal set of states C in the LTS of the system, the following holds: $\exists s \in C$, such that some action $a \in \{a_1, a_2 \dots a_n\}$ is enabled at s (we say that a is *enabled* in C). Similarly, a property “progress $P = \text{if } \{a_1, a_2 \dots a_n\} \text{ then } \{b_1, b_2 \dots b_n\}$ ” is satisfied iff in the LTS of the system, there is no terminal set of states where some action in $\{a_1, a_2 \dots a_n\}$ but no action in $\{b_1, b_2 \dots b_n\}$ are enabled.

The algorithm that decides whether a progress property is satisfied is therefore based on the computation of the terminal sets of states of a system. Terminal sets are found by computing the strongly connected components in the LTS graph and applying the additional criterion that no transition exists to a state outside the strongly connected component. Tarjan [29] showed that strongly connected components can be computed in linear time. Consequently, the check that progress properties hold is efficient. Note that it is only necessary to compute the terminal sets once to check any number of progress properties. As diagnostic information in case of progress violations, the LTSA tool displays a trace of actions leading to the terminal set together with the actions enabled in the set (see sample LTSA output above).

The LTSA performs a default progress check when no progress properties are explicitly specified. This consists of checking progress with respect to all actions in the alphabet of the system. For a system S , this is equivalent to checking that $\forall a \in \alpha S, \text{progress } P_a = \{a\}$. If no actions in αS are missing from terminal sets of states in S , then liveness is guaranteed in the system, since all actions always eventually occur. However, the liveness guarantee is with respect to the assumption of fair choice. We will see in the next section that liveness problems related to scheduling only become apparent when the system model is augmented to reflect *adverse* conditions.

3 Action Priority

The progress checking mechanism proposed in the previous section is based on the assumption of fair choice. This assumption corresponds to strong fairness on the system transitions, which is often too restrictive to be practical [27]. In fact, practical schedulers in computing systems do not implement fair choice [2]. This means that

some executions that may be exhibited by the system will be ignored by the checking mechanism as unfair. To find problems with such executions, we propose a simple action priority scheme that allows the user to “stress” a system by applying adverse scheduling conditions. With our scheme, a set of actions in a process is given higher or lower priority than the remaining ones in the process alphabet. We introduce the following abbreviations:

$$P \xrightarrow{a} \text{ to mean that } \exists P' \text{ such that } P \xrightarrow{a} P'$$

$$P \xrightarrow{b} \text{ to mean that } \exists P' \text{ such that } P \xrightarrow{a} P'$$

The low (high) priority operators \gg (\ll) take as arguments a process $P = \langle S_1, A_1, \Delta, q_1 \rangle$ and a set of actions $K \subseteq Act$, and return process $P \gg K = \langle S_1, A_1, \Delta, q_1 \rangle$ ($P \ll K = \langle S_1, A_1, \Delta, q_1 \rangle$), where the semantics for Δ are given by Rule 1 (Rule 2) below:

Rule 1: Let $a \in Act_\tau$. Then:

$$\frac{P \xrightarrow{a} P'}{P \gg K \xrightarrow{a} P' \gg K} \text{ if } ((a \notin K) \text{ or } (\forall b \in (A_1 - K), P \not\xrightarrow{b}))$$

Rule 2: Let $a \in Act_\tau$. Then:

$$\frac{P \xrightarrow{a} P'}{P \ll K \xrightarrow{a} P' \ll K} \text{ if } ((a \in K) \text{ or } (\forall b \in K, P \not\xrightarrow{b}))$$

Intuitively, $P \gg K$ expresses the fact that actions in K have lower priority than the remaining actions in αP . As a result, at any state where multiple actions are eligible, actions in K are ignored unless it is not possible to execute any action in $\alpha P - K$ instead. In contrast, in $P \ll K$, actions in K have high priority, so actions in $\alpha P - K$ are only selected when it is not possible to execute some action in K instead.

Action priority is thus used in our approach to force specific transitions to be taken when a choice is possible. Let P be the original system to be checked, and P' be the result of applying action priority to P . Then selected unfair executions of P will correspond to fair executions of P' . These unfair executions of P can therefore be checked with our mechanism by checking system P' under fair choice.

4 Example: Readers/Writers

To illustrate our approach to progress analysis using action priority, we will use the well-known Readers/Writers problem. This is concerned with access to a shared database by two kinds of processes. Readers execute transactions that examine the database while Writers both examine and update the database. For the database to be updated correctly, Writers must have exclusive access to the database while they are updating it. If no Writer is accessing the database, any number of Readers may concurrently access it. Access to the database is controlled by a read/write lock which processes must acquire before accessing the database. The FSP model for such a lock,

together with the processes that acquire and release it, is defined in Fig. 4. The system consists of the parallel composition of the user processes with the lock. The process *READWRITELOCK* is defined as a choice among a set of guarded actions controlled by the variables *writing* and *readers*. The action for a reader to acquire a lock is only permitted when *writing* is false indicating that the lock has not been acquired by a writer. The action for a writer to acquire the lock is only permitted when the lock has not been acquired for either read or write access (*readers==0 && !writing*). The LTS generated for the composition *READERS_WRITERS* is depicted in Fig. 5.

```

const Nread = 2           // Maximum readers
range R = 1..Nread
const Nwrite=2          // Maximum writers
range W = 1..Nwrite
range ReadR = 0..Nread
range WriteW = 0..Nwrite

READWRITELOCK = RW[0][False],
RW[readers:ReadR][writing:Bool] =
  (when (!writing && readers<Nread)
   reader[R].acquire -> RW[readers+1][writing]
 | when (readers>0)
   reader[R].release -> RW[readers-1][writing]
 | when (readers==0 && !writing)
   writer[W].acquire -> RW[readers][True]
 | when (writing)
   writer[W].release -> RW[readers][False]).

USER = (acquire -> release -> USER).

|| READERS_WRITERS =
  (reader[R]:USER || writer[W]:USER || READWRITELOCK).

```

Fig. 4. Readers/Writers system model

The progress properties of interest in this system are that writers can always acquire the lock and that readers can always acquire the lock. These properties can be specified as:

```

progress WRITER = {writer[W].acquire}
progress READER = {reader[R].acquire}

```

The progress property *WRITER* is satisfied if any *writer* in the range *W* acquires the lock. The property *READER* is satisfied if any *reader* in the range *R* acquires the lock. A progress check of these properties against the *READERS_WRITERS* system discovers no violations. Now we will examine the behaviour of the system under adverse conditions. For the *READERS_WRITERS* system, these adverse conditions occur when there is always competition for the lock. This happens when either the lock is requested frequently or the lock is held by processes for long periods. To model these conditions, we give release actions for both readers and writers lower priority than

acquire actions. Consequently, in any choice between acquiring and releasing the lock, acquiring it will have priority. This is described by:

```
||RW_PROGRESS = READERS_WRITERS
  >>{reader[R].release,writer[W].release}.
```

Progress analysis of this system results in the following violation:

```
Progress violation: WRITER
Trace to terminal set of states:
  reader.1.acquire
Actions in terminal set:
  {reader.1.acquire, reader.1.release,
   reader.2.acquire, reader.2.release}
```

This is the writer starvation situation in which writers do not get access because the number of readers with read access never drops to zero. In this simple example, the terminal set of states (3,4,5) causing the violation can be seen in the LTS of *RW_PROGRESS* depicted in Fig. 7.

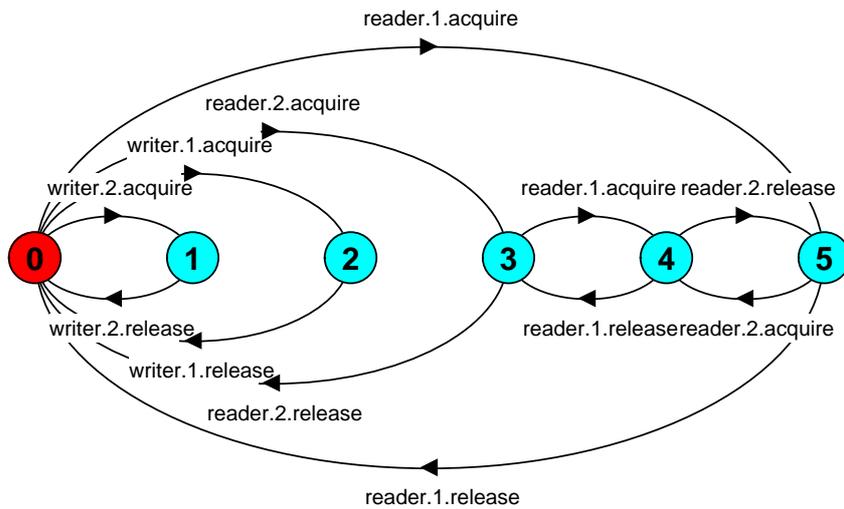


Fig. 5. LTS for *READERS_WRITERS*

The problem of writer starvation can be fixed by making readers defer to waiting writers. To detect waiting processes, we modify the definition of *USER* processes such that they request access to the lock before attempting to acquire it:

```
USER = (request-> acquire -> release -> USER).
```

The revised definition of the lock that uses this information is listed in Fig. 6. The new version keeps a count of waiting writers *ww*. Readers only acquire access if there are no writers waiting ($!writing \ \&\& \ readers < Nread \ \&\& \ ww == 0$). This new version of the lock when checked under the same conditions no longer detects a violation of the progress property *WRITER*. However, it is now possible for readers to starve:

```

Progress violation: READER
Trace to terminal set of states:
    reader.1.request
    reader.2.request
    writer.1.request
    writer.2.request
Actions in terminal set:
    {writer.1.request, writer.1.acquire,
     writer.1.release, writer.2.request,
     writer.2.acquire, writer.2.release}

```

```

READWRITELOCK = RW[0][False][0],
RW[readers:ReadR][writing:Bool][ww:WriteW] =
  (when (!writing && readers<Nread && ww==0)
   reader[R].acquire -> RW[readers+1][writing][ww]
  | when (readers>0)
   reader[R].release -> RW[readers-1][writing][ww]
  | when (readers==0 && !writing &&ww>0)
   writer[W].acquire -> RW[readers][True][ww-1]
  | when (writing)
   writer[W].release -> RW[readers][False][ww]
  | when (ww<Nwrite)
   writer[W].request -> RW[readers][writing][ww+1]
  | reader[R].request -> RW[readers][writing][ww]).

```

Fig. 6. Revised *READWRITELOCK*

The problem of reader starvation can of course be fixed by introducing a “turn” variable that lets readers and writers run alternately when competition exists for the lock. Such a system should satisfy both the *READER* and *WRITER* progress properties. Examples of conditional progress properties related to the *READERS_WRITERS* system are shown below:

```

progress WREL[i:W] =
    if {writer[i].acquire} then {writer[i].release}
progress RREL[i:R] =
    if {reader[i].acquire} then {reader[i].release}

```

The progress properties assert for each writer and for each reader that, if they regularly acquire the lock, they must also regularly release it. None of these properties is violated by the two versions of the system presented.

The checking mechanism that we have proposed has a number of advantages when compared to the approach based on Büchi automata. In the *READERS_WRITERS* example, each of the progress properties has to be checked separately if Büchi automata are to be used for verification. The Büchi automaton for the negation of property *WRITER* ($\Diamond \Box \neg (\text{writer.1.acquire} \vee \text{writer.2.acquire})$) is illustrated in Fig. 8. The transition *@WRITER* is used to mark the accepting state (1) of the automaton [6]. Note that when fair choice is assumed, a complete automaton must be used for verification. This is necessary since when a transition is undefined in the automaton, a non-terminal set of states may become terminal. A Büchi automaton can always be made complete by adding one state.

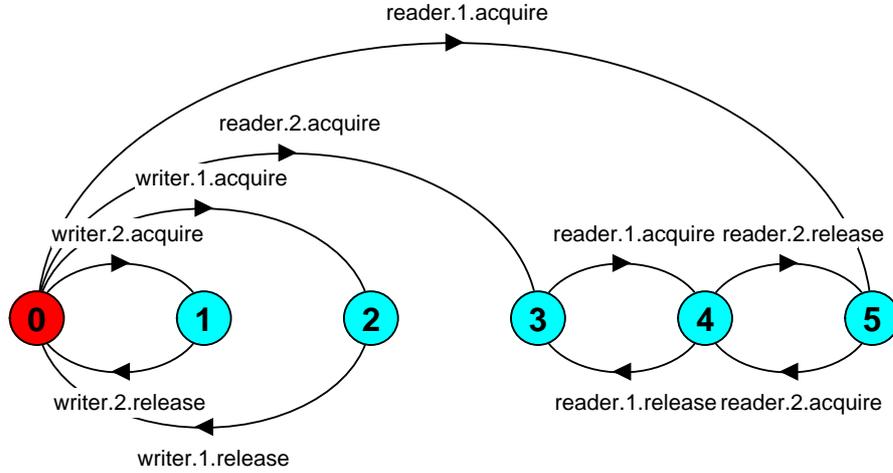


Fig. 7. LTS for *RW_PROGRESS*

The system *READERS_WRITERS* \parallel *WRITER* consists of 18 states. The size of the system has therefore increased by 3 times, which corresponds to the size of the Büchi automaton. For large systems, such an increase is significant. Additionally, in our approach, a single graph exploration is sufficient to check any number of progress properties, which is not the case with Büchi automata.

Finally, it should be noted that safety analysis must be performed on a system before action priority is applied for progress analysis purposes. Since action priority removes transitions, it may remove erroneous system behaviour.

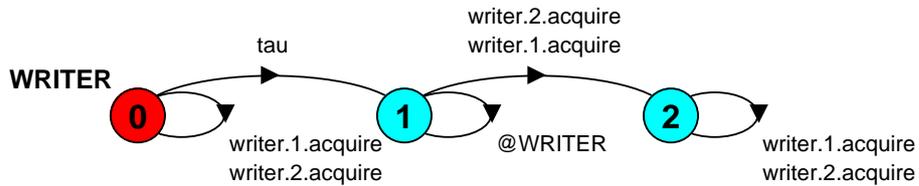


Fig. 8. Büchi automaton used for checking progress property *WRITER*

5 Related Work

Progress. Manna and Pnueli classify properties of programs into a hierarchy, where each class is characterised by a canonical temporal formula scheme [24]. They associate the term *progress* with several classes of this hierarchy. These formulas do not always correspond to liveness properties in the safety-liveness classification. Their work gives a detailed description of the differences between the two classifications. In fact, our progress properties are a subclass of the properties referred to in [24] as *re-*

sponse. The notion of progress also appears in Unity [5], where selected types of formulas are handled, and classified as safety and progress. Their progress properties correspond to LTL properties of the type $\Box(a \Rightarrow \Diamond b)$ (leads to) and aUb (ensures), where U denotes strong until.

SPIN [18] uses the notion of progress in a similar context to ours. The tool provides the facility to mark selected states of processes as progress states. It then checks that $\Box \Diamond progress$, where *progress* is true in a system state if at least one of the system processes is in a progress state. The SPIN liveness checks also incorporate a weak fairness assumption with respect to processes. The different fairness assumption and the fact that we specify progress in terms of actions rather than states are largely determined by the difference in analysis approaches. SPIN uses an on-the-fly approach to analysis, which preserves information about states in individual processes, whereas we use CRA, where this information is not preserved under composition.

Our approach differs significantly from that of SPIN both in terms of expressiveness, and algorithmically. Currently, SPIN performs progress checks by introducing a pre-defined Büchi automaton for progress. As a result, the state space of the system is affected (this also holds for the original algorithm presented in [18], where a two-state demon process was added to the model to determine different modes for the checking algorithm). Unlike our approach, SPIN's progress mechanism can deal neither with the conjunction of a number of progress properties, nor with conditional progress. For example, it cannot check if at least one progress state from *each* component process must occur infinitely often in the executions of a system. In SPIN, such properties are supported by the LTL model-checking mechanism. Further, we provide the option of action priority that permits a system to be checked under adverse scheduling conditions.

Fairness. The issue of fairness has been extensively investigated. Lehmann *et al.* introduced three notions of fairness that are useful in practice [21]. An infinite execution is *unconditionally* fair if every transition is taken infinitely often, *strongly* fair if for every transition, if it is enabled infinitely often it is executed infinitely often, and *weakly* fair if for every transition, if it is enabled continuously from some point on, it is taken infinitely often. The term transition can be substituted by process or action to obtain the same fairness conditions with respect to processes [8] or actions [20]. Weak, strong, and unconditional fairness are also referred to as justice, fairness (or compassion) and impartiality. Based on these definitions, our assumption of fair choice corresponds to strong fairness with respect to the system transitions. Different notions of fairness are appropriate for different system models. Apt *et al.* [3] present some criteria of effectiveness and utility of adopting some notion of fairness in a computational model.

Queille and Sifakis [27] stress the importance of defining fairness with respect to specific actions or predicates of the system, which they call relative fairness. Natarajan and Cleaveland [25] take such an approach, and propose a notion of weak fairness with respect to *success*, in order to determine when a process passes a test. The framework presented by Manna and Pnueli [24] supports the specification of weak and strong fairness with respect to specific system transitions.

A way of dealing with fairness in model checking is to add Büchi acceptance conditions to the system. For example in [1], all components of the system are Büchi automata, and therefore only executions that are acceptable by the product Büchi automaton are checked for correctness. Gribomont and Wolper [16] describe how a Büchi automaton can be used to express a fair process scheduler. Clarke *et al.* [8] extend their model with a set of predicates, so that fair paths are defined as paths in which each predicate holds infinitely often. This is equivalent to turning the model of the system into a generalised Büchi automaton. In this way, they can express both weak and unconditional fairness on processes. However, this requires the user to modify the initial model of the system. Finally, in Unity [5], the notion of fairness requires that every statement is selected infinitely often in any infinite execution.

Priority. Priority has been introduced as a means of assigning more importance to some actions than others. Examples of actions that require special treatment are interrupts and timeouts. In [26], Phillips performs a study and comparison between various approaches to introducing priority in process algebra. Relative vs. absolute and conditional vs. exclusive forms of priority appear in the literature. Recently, dynamic priority has also been proposed in the context of real-time systems [4]. In our approach, priority is not used as a modelling operator. Rather, it is simply used as a way of eliminating transitions, and obtaining system executions that would otherwise be considered unfair. Therefore, we do not need to consider whether the semantic equivalence of our model remains a congruence with the introduction of a priority scheme. As a result, we have taken a very simple approach to priority, similar to the initial one proposed by Cleaveland and Hennessy in [10].

6 Discussion and Conclusions

The work presented in this paper was motivated by a desire to achieve a balance between expressive power, accessibility and efficiency of analysis methods. Despite their expressive power, Büchi automata may exacerbate the state explosion problem. Moreover, they are not easy to specify without the use of an automated tool [19]. In general, this approach to verification is appropriate for experienced users of an analysis tool, that can use effectively a formalism like LTL or Büchi automata to specify properties or fairness assumptions of the system. The effort of using such a mechanism should only be required by the user if no simpler method is available for performing the specific analysis of interest.

In general, methods should require minimal effort before engineers start realising the benefits from their use [9]. The progress checking mechanism that we propose provides a way of checking liveness in a system, which is easily accessible by non-experts. Although less expressive than LTL and Büchi automata, progress properties can be specified in a simple intuitive way, and can be checked on the unmodified LTS of the system. In the context of CRA, progress properties are specified independently of the processes and composite subsystems that form a system. Consequently, they can be applied meaningfully to a subsystem as well as to the composite system as long as

the subsystem contains the progress actions in its alphabet. A single traversal of the LTS of a system is sufficient to check any number of progress properties.

In our framework, progress and safety properties can be combined efficiently, and checked simultaneously. Therefore, users need to revert to LTL model-checking only for restricted classes of liveness properties. Our experience so far in analysing architectural models leads us to believe that progress properties are sufficiently expressive to allow many liveness properties, of interest at the software architecture level, to be verified. For example, we applied our technique to a model of an Active Badge System with 566,820 states and 2,428,488 transitions [22], and showed that badge commands are not acknowledged if badges move between locations too frequently.

The combination of progress checks and action priority provides an elegant way of dealing with models that incorporate a notion of discrete time. The passing of time is modelled as a global tick action [28]. The maximal progress condition that is usually assumed for these discrete time models is ensured by making the tick action low priority: “>> {tick}”. The integrity of the model with respect to time can be checked by asserting the progress property “progress TIME = {tick}”. We have used these principles to construct and check a discrete time model for a Bounded Retransmission Protocol used in one of Philips’ products.

In their work on patterns in property specifications [11], Dwyer *et. al* report that the most common property pattern is *Response*, described in LTL as $\Box(a \Rightarrow \Diamond b)$. Our progress and conditional progress schemes cover a wide range of properties that fall in this category. For example, when $\Box \Diamond a$ holds in a system, $\Box(a \Rightarrow \Diamond b)$ reduces to the conditional progress property “progress Response = if {a} then {b}”.

The proposed fairness assumption has been elegantly incorporated in all our liveness-checking mechanisms [13] (though, in this paper, it was presented in the context of progress). We found that the notion of fairness with respect to *transitions* fits more naturally with our framework. In the context of CRA, it is not easy to apply fairness with respect to *processes* of the system, because the LTS of a composite system does not retain information about which processes it consists of. This could only be achieved by modifying the LTSs of the system components to record all necessary information, similarly to the approach proposed by Clarke *et al.* in [8].

In the context of liveness property checking, the possibility of including a notion of fairness is essential. When Büchi automata are used to express fairness constraints, users not familiar with the formalism are unable to check their model under any fairness conditions. In such cases, most of the counterexamples returned by the checking procedure correspond to unrealistic executions of the system analysed. As model checkers return a single counterexample for a property violation, the user has no way of finding out if the property checked is really violated, unless the counterexample is realistic. We believe that, rather than checking liveness with no fairness constraints and obtaining misleading violations, it is preferable from the developer’s point of view to get only realistic results from the tool, even at the risk of missing problems that may occur in practice.

The advantage of action priority is that it is simple to model, and the LTS of the system is automatically updated accordingly. The user can therefore easily experiment with checking various instances of the system behaviour, by applying different priori-

ties to it. As a result, the coverage of the checking mechanism under fair choice can be increased. This process is guided by users, who may enforce adverse scheduling conditions based on their intuition about vulnerable parts of the system behaviour.

In the context of CRA, action priority is applied to produce subsystem versions solely for checking progress at the subsystem level. These “test” subsystems are not used in constructing composite behaviours, since the application of action priority removes parts of system behaviour. In our implementation, action priority is applied during the construction of a composite LTS from component processes. Therefore, action priority can also be used for performing partial searches on systems that are too large for exhaustive exploration. In these cases, action priority provides a way of selecting interesting behaviours for analysis. The current priority scheme allows only coarse-grained control of scheduling. To refine this control, we plan to investigate the use of more powerful priority schemes, such as relative and dynamic action priorities.

Acknowledgements. We gratefully acknowledge the financial support provided by the EPSRC Grant GR/M24493 (BEADS project), and the ESPRIT LTR Project 24962 (C3DS). We also thank Iain Phillips for helpful discussions on action priority.

References

1. Aggarwal, S., Courcoubetis, C., and Wolper, P. “Adding Liveness Properties to Coupled Finite-State Machines,” *ACM Transactions on Programming Languages and Systems*, vol. 12(2), pp. 303-339, April 1990.
2. Andrews, G.R. *Concurrent Programming - Principles and Practice*: The Benjamin / Cummings Publishing Company Ltd, 1991.
3. Apt, K.R., Francez, N., and Katz, S. “Appraising fairness in languages for distributed programming,” *Distributed Computing*, vol. 2, pp. 226-241, 1988.
4. Bhat, G., Cleaveland, R., and Lüttgen, G. “Dynamic priorities for modeling real-time,” in *Proc. of the Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE X/PSTV XVII '97)*, Osaka, November 1997, pp. 321-336. T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, Eds.
5. Chandy, K.M. and Misra, J. *Parallel Program Design: a Foundation*: Addison-Wesley, 1988.
6. Cheung, S.C., Giannakopoulou, D., and Kramer, J. “Verification of Liveness Properties using Compositional Reachability Analysis,” in *Proc. of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'97)*, Zurich, Switzerland, September 1997. Lecture Notes in Computer Science 1301, pp. 227-243. M. Jazayeri and H. Schauer, Eds.
7. Cheung, S.C. and Kramer, J. “Checking Subsystem Safety Properties in Compositional Reachability Analysis,” in *Proc. of the 18th International Conference on Software Engineering (ICSE'18)*, Berlin, Germany, March 1996, pp. 144-154.
8. Clarke, E.M., Emerson, E.A., and Sistla, A.P. “Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications,” *ACM Transactions on Programming Languages and Systems*, vol. 8(2), pp. 244-263, 1986.
9. Clarke, E.M. and Wing, J.M. “Formal Methods: State of the Art and Future Directions,” *ACM Computing Surveys*, vol. 28(4), December 1996, pp. 626-643.
10. Cleaveland, R. and Hennessy, M. “Priorities in process algebra,” *Information and Computation*, vol. 87(1/2), pp. 58-77, July/August 1990.

11. Dwyer, M., Avrunin, G., and Corbett, J. "Patterns in property Specifications for Finite-State Verification," in *Proc. of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, 16-22 May 1999, pp. 411-420.
12. Gerth, R., Peled, D., Vardi, M.Y., and Wolper, P. "Simple On-the-fly Automatic Verification of Linear Temporal Logic," in *Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95)*, Warsaw, Poland, June 1995, pp. 3-18.
13. Giannakopoulou, D. "Model Checking for Concurrent Software Architectures," PhD Thesis, Dept. of Computing, Imperial College, London, March 1999.
14. Giannakopoulou, D., Kramer, J., and Cheung, S.C. "Analysing the Behaviour of Distributed Systems using Tracta," *Journal of Automated Software Engineering, special issue on Automated Analysis of Software*, vol. 6(1), January 1999. R. Cleaveland and D. Jackson, Eds.
15. Godefroid, P. and Holzmann, G.J. "On the Verification of Temporal Properties," in *Proc. of the 13th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification (PSTV'93)*, Liège, Belgium, June 1993, pp. 109-124. A. Danthine, G. Leduc, and P. Wolper, Eds.
16. Gribomont, P. and Wolper, P. "Temporal Logic," in *From Modal Logic to Deductive Databases*, A. Thayse, Ed.: John Wiley and Sons, 1989.
17. Hoare, C.A.R. *Communicating Sequential Processes*: Prentice-Hall, 1985.
18. Holzmann, G.J. *Design and Validation of Computer Protocols*: Prentice Hall, 1991.
19. Holzmann, G.J. "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23(5), pp. 279-295, May 1997.
20. Lamport, L. "The Temporal Logic of Actions," *ACM Transactions on Programming Languages and Systems*, vol. 16(3), pp. 872-923, May 1994.
21. Lehmann, D., Pnueli, A., and Stavi, J. "Impartiality, Justice and Fairness: The ethics of concurrent termination," in *Proc. of the 8th International Colloquium on Automata, Languages and Programming*, Acre (Akko), Israel, July 13-17, 1981. Lecture Notes in Computer Science 115, pp. 264-277. S. Even and O. Kariv, Eds.
22. Magee, J., Kramer, J., and Giannakopoulou, D. "Analysing the Behaviour of Distributed Software Architectures: a Case Study," in *Proc. of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, Tunis, Tunisia, October 1997, pp. 240-245.
23. Magee, J., Kramer, J., and Giannakopoulou, D. "Software Architecture Directed Behaviour Analysis," in *Proc. of the Ninth IEEE International Workshop on Software Specification and Design (IWSSD-9)*, Ise-shima, Japan, April 16-18 1998, pp. 144-146.
24. Manna, Z. and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems - Specification*: Springer-Verlag, 1992.
25. Natarajan, V. and Cleaveland, R. "Divergence and Fair Testing," in *Proc. of the Automata, Languages and Programming (ICALP '95)*, Szeged, Hungary, July 1995. Lecture Notes in Computer Science 944, pp. 648-659. Z. Fulop and F. Gecseg, Eds.
26. Phillips, I. "Approaches to priority in process algebra," Draft Report, Dept. of Computing, Imperial College, London, November 1994.
27. Queille, J.P. and Sifakis, J. "Fairness and Related Properties in Transition Systems - A Temporal Logic to Deal with Fairness," *Acta Informatica*, vol. 19, pp. 195-220, 1983.
28. Roscoe, A.W. *The Theory and Practice of Concurrency*: Prentice Hall, 1998.
29. Tarjan, R. "Depth-First Search and Linear Graph Algorithms," *SIAM Journal of Computing*, vol. 1, pp. 146-160, 1972.
30. Vardi, M.Y. and Wolper, P. "An automata-theoretic approach to automatic program verification," in *Proc. of the 1st Symposium on Logic in Computer Science*, Cambridge, June 1986, pp. 322-331.