Fairness and Priority in Progress Property Analysis*

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
{dg1, 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 modelling explicitly 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. An extensive comparison is provided of the approach proposed with classical LTL model-checking.

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 composite model of the system based on the models of its components [1-3]. 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 [4, 5]. Our work on liveness property checking [4, 6] takes the automata-theoretic approach to verification [7], adopted in a number of existing methods and tools [8-10]. 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 viewed as 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, the property F is not violated. In such a setting, fairness is represented in terms of constraints introduced in the form of Büchi automata, which are also composed with the system [8, 9]. Alternatively, in order to check some LTL formula f under some LTL fairness condition c, one can simply check the LTL formula $(c \Rightarrow f)$ [10].

In general, it is a challenging task to think of the exact fairness considerations that need to be made for a system. Often, one finds that the constraints imposed are either not sufficient, or too strict, and that, as a result, unrealistic violations are detected, or some violations that may

* This article is a revised and extended version of a paper presented at the 7th European Software Engineering Conference, held jointly with the 7th ACM SIFSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'99), September 1999.

in practice occur are not detected, respectively. Our experience is that the fairness assumptions that apply to a system are determined gradually. Users check properties under fairness assumptions that they may tailor, based on the feedback obtained from model checkers. For this reason, we consider it important for a model-checking tool to support predefined notions of fairness, and to provide the flexibility to easily tune these fairness assumptions to express different execution conditions of the system.

In accordance with these requirements, this paper proposes an optional predefined assumption of "fair choice" on the executions of an LTS model. As this is a strong assumption that may be too restrictive in some circumstances, we also introduce an action priority scheme that relaxes it. This combination provides a simple, practical and effective way of expressing and tailoring fairness assumptions during liveness property checks. Moreover, we have found that under fair choice, a specific class of liveness properties, which we have termed progress, can be checked without using Büchi automata. We express such properties in an intuitive way that does not require experience with temporal logic.

In addition to its simplicity and flexibility, the overall approach that we present does not increase the state-space of a system to be analysed. It is well known that, when a Büchi automaton *B* is composed with a system, the size of the system can 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 [8]. Although efficient algorithms exist for the automatic translation of LTL formulas into Büchi automata [11], none of these algorithms guarantee to generate the minimal automaton. Of course, the potential advantage from the use of Büchi automata is that in the best case, the state-space to be analysed is *zero*. This happens when no initial portion of the behaviour accepted by the automaton appears in the system. However, we will show that this best case does not normally appear for the liveness properties and fairness constraints that we consider.

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 [12] indicate that a large number of interesting properties of systems can be expressed and checked in terms of progress properties.

The results of our work have been incorporated in an analysis tool – the Labelled Transition System Analyser (LTSA) [2, 13]. 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 <u>F</u>inite <u>S</u>tate <u>P</u>rocesses, 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. In the interests of brevity, we do not formally describe the FSP semantics in this paper. Rather, we hope to make the meaning of our specifications clear by means of brief descriptions, and by including the associated LTSs, when appropriate.

The remainder of this paper is organised as follows. The next section uses a simple telephone switching system to introduce FSP. The example is used again in section 3, to describe informally the way progress properties are specified and checked under the proposed fairness and action priority schemes. Section 4 provides the formal framework for our approach. Section 5 presents the Readers/Writers example that is used to illustrate and evaluate our approach. In section 6, we look into the classical way of checking LTL properties with Büchi automata, and compare our approach with that taken by the SPIN LTL model checker. Our experience with several case studies is discussed in section 7. Finally, section 8 presents related work, and section 9 closes the paper with conclusions and plans for future work.

2 A Telephone Switching System

A simple telephone switching system will be used as an introductory example to both our specification language FSP, and to our approach to progress and fairness.

```
const Phones = 2

PHONE (Id = 0) = (call[Id] ->
                    ( connect[Id] -> end[Id] -> PHONE
                    | engaged[Id] -> PHONE) ).

SWITCH = (call[i:1..Phones] -> connect[i] -> ENGAGED[i]),
ENGAGED[i:1..Phones]=( call[x:1..Phones] -> engaged[x] -> ENGAGED[i]
                     | end[i] -> SWITCH).

|| PHONES = (forall [i:1..Phones] PHONE(i) || SWITCH).
```

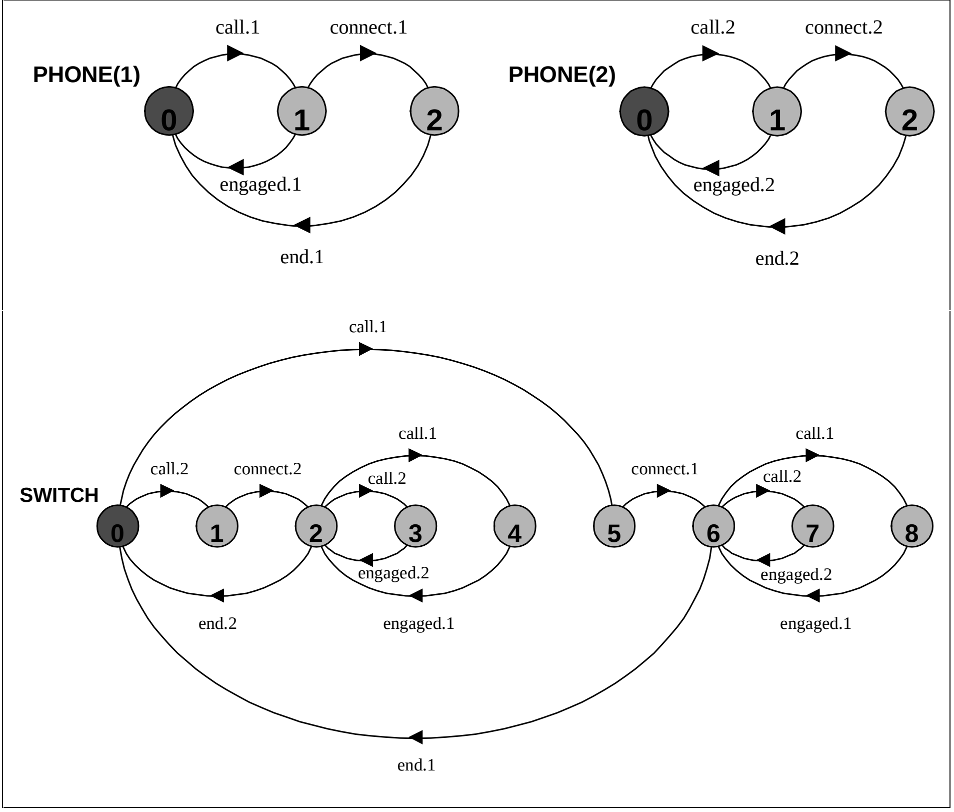


Fig. 1. FSP specification and corresponding LTSs for telephone switching system

The system, whose FSP description is provided in Fig. 1, consists of a *SWITCH* that handles incoming calls to some recipient (let us call it *Rec*), and a number of *PHONE*s in range *Phones*, from which these calls may originate. The behaviours of *SWITCH* and *PHONE* are defined using action prefix ("->"), choice ("|") and recursion. On receipt of a call from some phone *i* (*call*[*i:1..Phones*]), the switch connects *i* to *Rec* (*connect*[*i*]) and transits to state *ENGAGED*[*i*], which represents the fact that *Rec* is currently engaged. If further calls are received while the switch is in state *ENGAGED*, then the switch notifies their originators that the line is *engaged*. On completion of a telephone conversation (*end*[*i*]), the switch goes back to its initial state. A phone is characterised by its *Id* (such parameters are given default values in FSP, which can be overridden in composition expressions). It is used to make a call to *Rec* (*call*[*Id*]), after which one of two alternative behaviours is possible. The switch connects the phone and after the conversation *end*s the *PHONE* goes back to its initial state. Alternatively, if the line is *engaged*, *PHONE* simply returns to its initial state. The LTSs generated by our tools for the components of the telephone switching system are illustrated in Fig. 1.

The *PHONES* system is expressed as the parallel composition of *SWITCH*, with as many instances of *PHONE* as the constant *Phones* defines (in our example, two). Processes assembled with the || parallel composition operator run concurrently by synchronisation on actions that are common to their alphabets and interleaving of the remaining actions. Based on these rules, the LTS that corresponds to system *PHONES* is as depicted in Fig. 2.

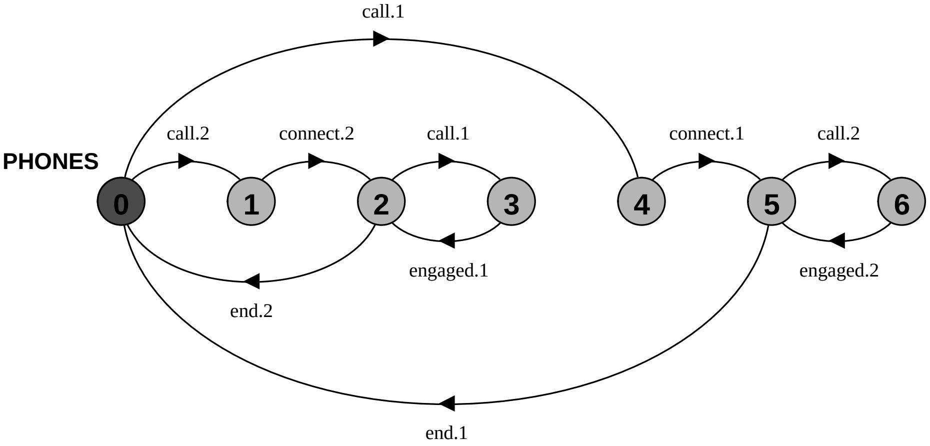


Fig. 2. LTS for the composite telephone switching system

3 Progress, Fairness, and Priority

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

The LTL formulation of the progress property *P* is $\Box\Diamond(a_1 \vee\ a_2 \ldots \vee a_n)$.

For system *PHONES* of the previous section, it is likely that a designer would expect both progress properties *GET_THROUGH.1* and *GET_THROUGH.2* to hold, where[1]:

```
progress GET_THROUGH.1 = {connect[1]}
progress GET_THROUGH.2 = {connect[2]}
```

The reason is that an execution of the system where calls from a particular phone are never connected is clearly undesirable. When thinking about progress properties, one implicitly makes some fairness assumptions about the system they apply to. For example, consider the LTS of *PHONES*, illustrated in Fig. 2. At state (0), transitions (0, *call.1*, 4) and (0, *call.2*, 1) are enabled. Assuming that the system scheduler does not favour one or the other transition, one would expect that, if the system returns to state (0) regularly, both transitions will be selected regularly. However, the LTS model is not expressive enough to capture such notions of fairness. In fact, it is possible for the LTS of Fig.2 to repetitively go through states (0), (1) and (2), indefinitely. For this reason, we introduce in the LTS model the notion of *fair choice*, which is assumed to hold when we check progress properties of a system:

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

Under fair choice, progress properties *GET_THROUGH*[*i*] hold for *PHONES*. Now consider the case where the second phone may get barred because, for example, its owner has not paid a telephone bill. The behaviour of such a *B_PHONE*, and of a system that includes it could be modelled as follows:

```
B_PHONE (Id = 0) =  ( call[Id] -> ( connect[Id] -> end[Id] -> B_PHONE
                                  | engaged[Id] -> B_PHONE)
                    | barred[Id] -> STOP).

|| B_PHONES = ( forall [i:1..Phones-1] PHONE(i)
              || B_PHONE(Phones) || SWITCH).
```

For simplicity, we assume that a phone does not get barred while used for a call. The LTS of system *B_PHONES* is illustrated in Fig. 3. We can see that in this system, progress property *GET_THROUGH.2* is no longer satisfied: action *connect.2* can only occur finitely many times in any *fair* infinite execution that reaches states (1), (2), or (3) at some point. The set of states {1,2,3} 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 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. The LTSA reports the violation as follows:

```
Progress violation: GET_THROUGH.2
Trace to terminal set of states:
      call.1
      connect.1
      barred.2
Actions in terminal set:
{call.1, connect.1, end.1}
```

[1] In FSP, such combinations of properties can be expressed by using ranges as follows:

```
progress GET_THROUGH[i:1..Phones] = {connect[i]}
```

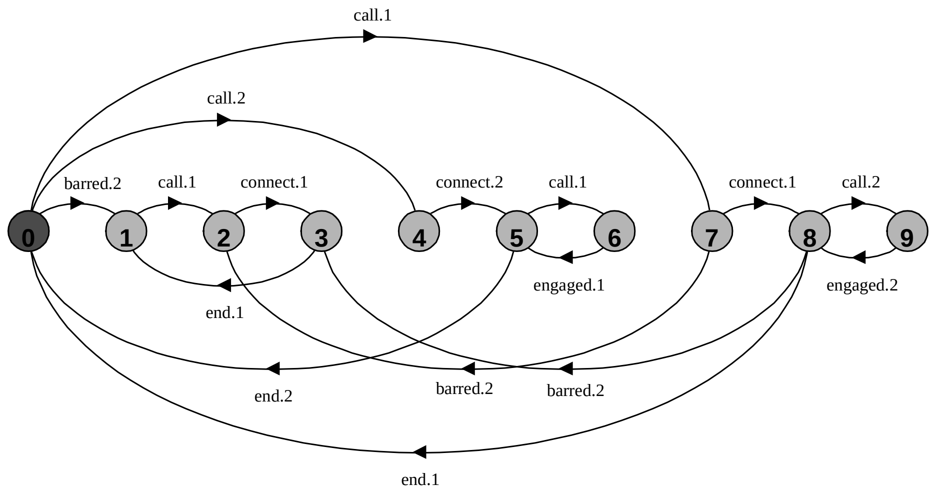


Fig. 3. LTS of system *B_PHONES*, where the second phone may be barred

However, this violation does not correspond to a real problem with the system. It is obvious that *connect.2* cannot occur infinitely often if, after some point, *call.2* no longer occurs either. So the desired property is in fact that, if calls are made from a specific client regularly, then this client must also be connected regularly, i.e., for every client *i*, the following LTL property must hold: $\Box\Diamond call.i \Rightarrow \Box\Diamond connect.i$. We call this form of progress property *conditional progress*, which we define as follows:

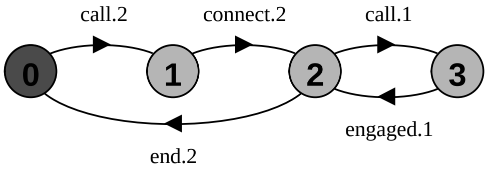
> **progress** P = **if** {a_1,a_2..a_n} **then** {b_1,b_2..b_n} – defines a progress property P which asserts that in any infinite execution of a target system, if any of the actions a_1,a_2..a_n occurs infinitely often then at least one of the actions b_1,b_2..b_n also occurs infinitely often.

Progress properties *GET_THROUGH* can therefore be restated as follows:

```
progress GET_THROUGH[i:1..Phones] = if {call[i]} then {connect[i]}
```

This property is satisfied by system *B_PHONES*; after the second phone is barred, calls are no longer made from it. The property therefore ensures that, when a phone is operational, calls made from it are not consistently ignored, which is what the user wishes to check.

Assume now that we wish to check whether the above progress properties are still satisfied in the original *PHONES* system, when the scheduler favours calls made from *PHONE(2)*. Such a *PRIORITY_PHONES* system is modelled, in our approach, by giving high priority to action *call.2*, i.e. to calls received from *PHONE(2)*. The FSP model for this system, together with its corresponding LTS are illustrated below:



```
|| PRIORITY_PHONES = PHONES << {call[2]}.
```

What has actually happened to the LTS of *PHONES* (Fig.2) by giving high priority to action *call.2* is that, at state (0), where both *call.1* and *call.2* are available, *call.2* is always selected, which results in cutting off the part of the LTS that involves states (4), (5) and (6). If we now check the *GET_THROUGH* progress properties on this system, we obtain the following result:

```
Progress violation: GET_THROUGH.1
Trace to terminal set of states:
Actions in terminal set:
{call.1, engaged.1, call.2, connect.2, end.2}
```

Clearly, with such a scheduler, *PHONE(1)* is never connected, because when *PHONES* is in state (0) (see Fig. 2), transition (0, *call.2*, 1) is always the one selected. Note that, if fair choice had not been assumed in the original *PHONES* system, then both progress properties *GET_THROUGH.1* and *GET_THROUGH.2* would be violated. A counterexample for property *GET_THROUGH.2* would then be the one obtained for system *PRIORITY_PHONES*. However, as discussed, such violations are in general not interesting to the user, since any reasonable scheduler implements some sort of fairness. For this reason, our approach is based on checking progress under fair choice, and then *selectively* stress-checking the system by applying action priority.

4 Formal Framework

This section formally presents our approach to analysing progress properties. It also introduces the notion of action priority, and its role in counterbalancing the effects of fair choice.

4.1 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 $s \in S$, if $\exists\ s' \in S$ such that $(s, a, s') \in \Delta$. Similarly, we say that a transition $(s, a, s') \in \Delta$ is enabled at a state $t \in S$ if $t = s$.

We call an *execution* of P an infinite sequence $q_0 a_0 q_1 a_1 \ldots$ 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 [14].

A state r is *reachable* from a state s in an LTS $P = \langle S, A, \Delta, q\rangle$, if (($r = s$) or ($\exists\ a \in A$ and $t \in S$, such that $(s, a, t) \in \Delta$ and r 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*$(s, P) = \{r \in S \mid r$ is reachable from s in $P\}$. An LTS $P = \langle S, A, \Delta, q\rangle$ transits into an LTS $P' = \langle S, A, \Delta, q' \rangle$ with an action $a \in A$ if $(q, a, q') \in \Delta$. That is, $\langle S, A, \Delta, q\rangle \xrightarrow{a} \langle S, A, \Delta, q' \rangle$ if $(q, a, q') \in \Delta$.

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 if $\forall s \in C$, *Reachable*$(s, P) = C$.

4.2 Checking Progress under Fair Choice

Our progress-checking algorithm is based on the following theorem:

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*.

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 $\forall s \in S_1$, $Reachable(s,P) \subseteq S_1$. 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 $\forall s \in S_1$, $S_1 \subseteq Reachable(s,P)$. We conclude that $\forall s \in S_1$, $Reachable(s,P) = S_1$ and therefore S_1 is a terminal set of states. ■

The Terminal Set Theorem establishes 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`..`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`..`a_n`}` is enabled at *s* (we say that *a* is *enabled* in *C*). Similarly, a property "`progress P = if {`a_1`,`a_2`..`a_n`} then {`b_1`,`b_2`..`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`..`a_n`}` but no action in `{`b_1`,`b_2`..`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 [15] 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 $\forall\ a \in \alpha S$, `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.

4.3 Action Priority

The progress checking mechanism proposed is based on the assumption of fair choice, which corresponds to strong fairness on the transitions of the *global* system, i.e., of the LTS obtained by the parallel composition of the system processes. This is a strong condition; for example, it is stronger than the classic notion of strong fairness with respect to the system processes (this notion is explained in section 8). However, even strong fairness is in general too restrictive to be practical [16]. In fact, practical schedulers in computing systems do not implement strong fairness [17]. 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-test 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:

$P \xrightarrow{a}$ to mean that $\exists\, P'$ such that $P \xrightarrow{a} P'$
$P \not\xrightarrow{a}$ to mean that $\nexists\, P'$ such that $P \xrightarrow{a} P'$

The *low* (*high*) *priority* operators >> (<<) take as arguments a process $P = \langle S_1, A_1, \Delta_1, q_1\rangle$ and a set of actions $K \subseteq Act$, and return process $P{>>}K = \langle S_1, A_1, \Delta, q_1\rangle$ ($P{<<}K = \langle S_1, A_1, \Delta, q_1\rangle$), where the semantics for Δ are given by Rule 1 (Rule 2) below:

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

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

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

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

Intuitively, $P{>>}K$ 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 $\alpha P\text{-}K$ instead. In contrast, in $P{<<}K$, actions in K have high priority, so actions in $\alpha P\text{-}K$ 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. It should be mentioned that fairness and action priority are only applicable in the context of liveness-property checking. Safety analysis is always carried out on the original system because, since action priority removes transitions, it may remove erroneous system behaviour. This is in lines with our methodology where, as described in section 7, safety analysis is performed first.

5 Case Study: 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 and the processes that acquire and release it, is defined below.

```
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)
                        reader[R].acquire -> RW[readers+1][writing]
                    | reader[R].release -> RW[readers-1][writing]
                    | when (readers==0 && !writing)
                        writer[W].acquire -> RW[readers][True]
                    | writer[W].release -> RW[readers][False]).

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

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

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. 4.

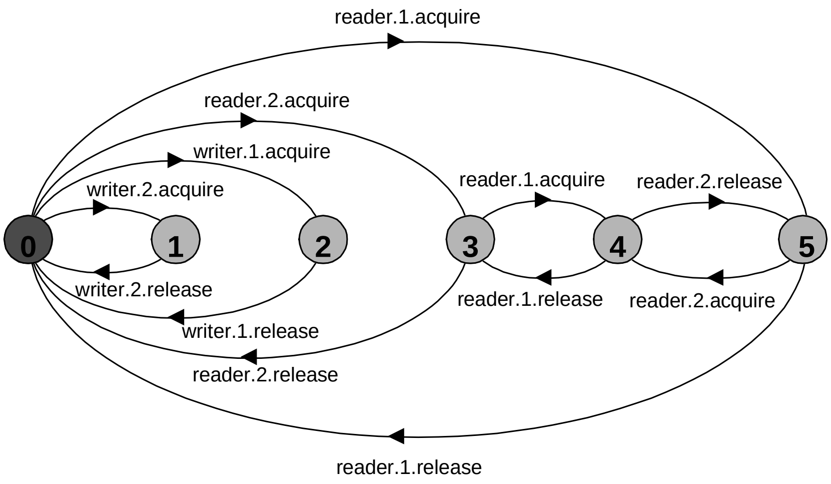


Fig. 4. LTS for *READERS_WRITERS*

The progress properties of interest in this system are that writers can always acquire the lock and that readers can always acquire the lock:

```
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 low priority. Consequently, in any choice between acquiring and releasing the lock, acquiring it will have priority. On the other hand, the same priority is given to the acquire actions of readers and writers. This reflects our assumption that the system will be running under a scheduler that is not consistently biased against one or the other type of process, when both types are interested in acquiring the lock. The system we check thus becomes:

```
||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. 5.

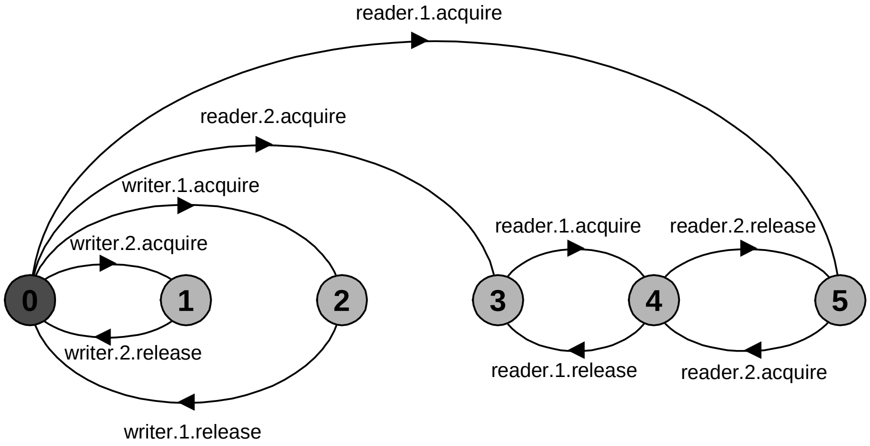


Fig. 5. LTS for *RW_PROGRESS*

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 below:

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

The new version keeps a count of waiting writers *wwaiting*. Readers only acquire access if there are no writers waiting (!*writing* && *readers*<*Nread* && *wwaiting* ==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}
```

The problem of reader starvation can be fixed by introducing a “turn” variable that lets readers and writers run alternately when competition exists for the lock, as follows:

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

The above system satisfies 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}
```

These 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.

6 Checking Progress Properties with Büchi Automata

As discussed, one of the motivations for our approach to progress and fairness is that the use of Büchi automata may exacerbate state explosion. This section explains how this problem occurs, and demonstrates it in terms of the Readers/Writers problem in the context of the LTSA and SPIN model checkers. Büchi automata are *finite* automata on *infinite* inputs and can, as such, be used to specify liveness properties of a system.

Definition – A Büchi automaton B is a 5-tuple $\langle S, A, \Delta, q_0, F\rangle$, where S is a finite set of states, $A \subseteq Act_\tau$ is a set of actions, $\Delta \subseteq S \times A \times S$ is a set of transitions on observable actions, $q_0 \in S$ is the initial state, and $F \subseteq S$ is a set of accepting states. ■

An *execution* of $B = \langle S, A, \Delta, q_0, F\rangle$ on an infinite word $w = a_0 a_1 a_2 \ldots$ over A is an infinite sequence $\sigma = q_0 a_0 q_1 a_1 q_2 \ldots$, where $(q_i, a_i, q_{i+1}) \in \Delta$, $\forall i \geq 0$. An execution σ is *accepting* if it contains some accepting state of B an infinite number of times. As Büchi automata may be non-deterministic, there can be several alternative executions of an automaton on a given infinite word. An infinite word w is *accepted* by B if there exists an accepting execution of B on w.

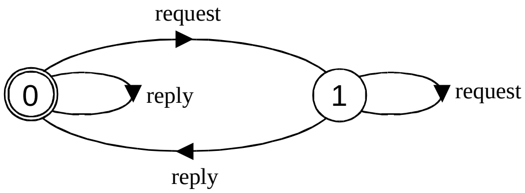


Fig. 6. Büchi automaton for property □(request ⇒ ◊ reply)

For example, the automaton illustrated in Fig. 6 accepts all words over {*request*, *reply*} which satisfy the property □(*request* ⇒ ◊ *reply*). The accepting state (0) is distinguished by the double circle. The automaton accepts any infinite word that can take it through state (0) an infinite number of times. Therefore, it will never accept a word which, at some point, contains a *request* that is never followed by a *reply*.

A Büchi automaton is called *empty* if it does not accept any word over its alphabet. A Büchi automaton is non-empty if at least one cycle in its graph contains some accepting state [8]. This can be explained intuitively as follows. For finite-state systems, an infinite execution can be obtained by following a path to some state *r* of some cycle, and then indefinitely following the path from *r* to *r* defined by the cycle. If the cycle contains some accepting state, then the execution is accepting.

Büchi automata are strictly more expressive than LTL. More specifically, any LTL formula can be algorithmically translated into a Büchi automaton that accepts exactly those infinite words over its alphabet that satisfy the formula [7]. Based on this fact, a widely used method for checking that a finite-state process *P* satisfies an LTL formula *F* proceeds as follows:

1. Build the Büchi automaton for $\neg F$.
2. Build the automaton corresponding to the *intersection* of the automaton describing the program with the automaton obtained for $\neg F$.
3. Check if the automaton obtained in step 2 is empty.

The first step of the above procedure can be performed algorithmically: an efficient algorithm has been proposed by [11]. However, as proven by Gribomont and Wolper in [8], the size of a Büchi automaton may increase exponentially as a function of the length of the formula that it represents (the length of the formula being the number of literals and operators in it). Moreover, there is no guarantee that the algorithm yields the minimal automaton corresponding to the formula. This fact becomes apparent in section 6.2, in the context of SPIN.

Step 2 computes the intersection of the system with the automaton generated by step 1. The state-space of the intersection of the system with the automaton is equal, in the worst case, to the product of their respective state-spaces. Therefore, a property may increase the number of the states that need to be explored during verification. In the best case, the resulting state-space is *zero*. This happens when no initial portion of the behaviour accepted by the automaton appears in the system. However, this best case normally appears when the formula being checked is a safety property. For purely liveness properties, it is not possible to decide, after exploring some finite initial portion of an execution, that the execution does not satisfy the property. This is due to the fact that any arbitrary finite sequence of actions can be extended to an infinite sequence satisfying a specific liveness property [18]. In sections 6.1 and 6.2, we show that progress properties indeed increase the state-space of a system, when expressed as Büchi automata.

Finally, step 3 can be performed by computing the strongly-connected components in the graph of the Büchi automaton generated from step 2, for which there exist efficient algorithms (the one in [19] for example is linear to the size of the graph).

The above approach to verification is adopted in a number of existing methods and tools [8-10]. In [6], we show how we adapted this basic mechanism to fit our CRA framework. In our approach, Büchi automata are introduced as simple components of a system described as a hierarchy of components. The intersection of an LTS with a Büchi automaton is computed as a parallel composition between LTSs.

Note that, the use of non-deterministic Büchi automata in checking properties of a system restricts the user to check one property at a time. Of course, one could claim that the conjunction of all the desired properties could be checked as a single property. However, this is not recommended, due to the fact that lengthier properties normally result in larger Büchi automata. This is also demonstrated in section 6.2. Checking properties one at a time introduces the following inconvenience: when changes are introduced in a system to correct some part of its behaviour that violates a specific property, all properties that have already been checked must be checked again, one by one. This is obviously avoided by our progress-checking mechanism, since any number of progress properties can be checked *simultaneously* on the *unmodified* (since no property automata are required) state-space of the system.

6.1 LTSA

In this section, we show how properties of the Readers/Writers problem can be checked with Büchi automata, in the context of the LTSA tool. LTSA is not yet equipped with an LTL translator, and therefore the automata used have been generated manually. For example, we used automaton *WRITER*, illustrated at the bottom of Fig. 7, to check progress property *WRITER*. Automata used for verification correspond to negated properties. Therefore automaton *WRITER* expresses the LTL formula ($\Diamond\Box\neg$(*writer.1.acquire* $\vee$ *writer.2.acquire*)).

As described in [4], we mark accepting states of an automaton with a special transition, which we call *accepting* transition. Such transitions loop from an accepting state to itself, and are named by prefixing the name of the automaton by the character @, reserved to denote an accepting transition. According to this, state (1) is the accepting state in both automata illustrated in Fig. 7. The intersection of an automaton with the system is then computed as an LTS parallel composition [6].

Intuitively, the automaton at the top of Fig. 7 accepts any execution where, at some *random* point, actions *write.1.acquire* and *write.2.acquire* never occur again. This random point is reflected by state (1), which is an accepting state, and where the automaton may jump non-deterministically (by performing action τ). The non-deterministic step is necessary to express the fact that this state need only be reached *eventually*.

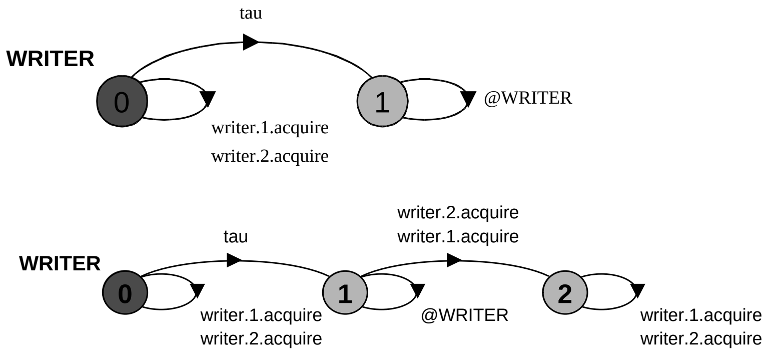


Fig. 7. ***(top)*** **Büchi automaton for property *WRITER* – *(bottom) WRITER* made complete**

An automaton is *complete* if, at every state, all the actions in its alphabet are enabled. In order to check the property under fair choice, we need to use a complete version of automaton *WRITER*. The reason is that, with fair choice, we only need to check *terminal sets of states* in the graph of the system. An undefined transition in the Büchi automaton that is composed with the system may cause a non-terminal set of states to become terminal. This is illustrated in Fig. 8, where *SYS_WRITER* corresponds to the intersection of system *SYS*, with the automaton *WRITER* at the top of Fig. 7. Although it is obvious that *SYS* does not violate the desired property under fair choice, *SYS_WRITER* contains a terminal set of states, {1}, where the accepting transition is enabled. The misleading violation is due to the fact that the automaton used was not complete.

A Büchi automaton can always be made complete by adding one state. The added state is a non-accepting state, and undefined transitions of the initial automaton are made to lead to the new state. For example, automaton *WRITER* at the top of Fig. 7 is transformed into the one at the bottom of the same figure. The new state is state (2), and the added transitions are transitions: (1, *writer.1.acquire*, 2) and (1, *writer.2.acquire*, 2).

Using the complete automaton, the system *READERS_WRITERS* || *WRITER* consists of 18 states. This system is larger than the original *READERS_WRITERS* by 3 times, which corresponds to the size of the Büchi automaton. It is clear that, when complete automata are used for verification, the system state-space is always increased as a result of computing its intersection with the automaton of interest.

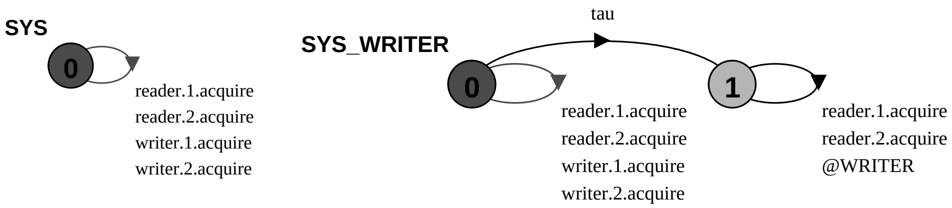


Fig. 8. A complete Büchi automaton must be used when fair choice is assumed

6.2 SPIN

SPIN [20] performs model checking on-the-fly, which means that it checks the state-space of the system while generating it. This is combined with efficient partial-order reduction techniques, as a means of avoiding state-explosion. The advantage of on-the-fly techniques is that they can stop at the first error encountered, rather than exploring the entire state-space of a system. These techniques are ideal for early stages of design, which tend to contain many errors. However, when a system is correct, such model checkers still need to explore the entire state-space.

In SPIN, liveness properties can be checked under the optional constraint of weak fairness with respect to the system processes. Under this constraint, every process that contains at least one transition that remains continuously enabled will execute within finite time. Weak fairness is implemented using Choueka's flag construction algorithm [21]. The exploration is then performed on *n*+1 copies of the state-space, where *n* is the number of processes in the system. This means that, theoretically, SPIN's weak fairness option increases the state-space to be explored by *n*+1 times, in the worst case. Moreover, SPIN's cycle detection algorithm is based on a nested depth-first search, which theoretically also doubles the system state-space. However, the designers of the tool report that: " *The worst case additional complexity contributed by this construction is a multiplication of the CPU time requirements by 2 x (n+1), where n is the number of processes in the Promela model. The memory requirements remain largely unaffected, by virtue of the storage technique that is explained in [10]* " (http://cm.bell-labs.com/cm/cs/what/spin/Man/WhatsNew.html).

Readers/Writers: Version 1

The input language of SPIN, Promela, is different in style than FSP. Therefore, the aim of this section is not to compare the state-spaces generated by Promela and FSP for the *READERS/WRITERS* problem. Rather, we intend to illustrate, in the context of SPIN – one of the most efficient LTL model checkers – the problems related with the use of Büchi automata in model checking. The Promela models discussed in this section are included in the Appendix. We only describe them briefly here – they should in general be straightforward.

We define two types of processes in Promela, *Reader* and *Writer*. The keyword active that is prefixed to the proctype definitions, together with the array suffixes *[R]* and *[W]*, are used to specify that *[R]* processes of type *Reader* and *[W]* processes of type *Writer* are active in the initial system state. The Read/Write lock is implemented by means of two global variables: *readers* counts the number of readers currently accessing the database, and *writing* records whether any writer is currently accessing the database. There is no need to have a separate process for the Read/Write lock, because these variables can be tested and set with an *atomic* step, as follows:

```
atomic {       /* acquire read */
      (!writing && readers <= R) -> readers++;
}
```

In SPIN, LTL formulas are *state-based* rather than *action-based.* To check properties related to readers and writers accessing the database, we label the corresponding states in the process definitions with labels *RD* and *WT*, respectively. In this way, property *WRITER* is expressed as $\Box\Diamond a$, where *a* is defined as follows:

```
#define a  (Writer[2]@WT || Writer[3]@WT)
```

Note that *Writer[i]@WT* means that the writer with process id *i*[2] is currently at its local state labelled with *WT*. We use the SPIN LTL translator to automatically translate the property into a Promela never-claim, which we include in the specification.

Initially, we check the Promela specification for the first version of Readers/Writers against safety (i.e., assertions and deadlocks). The following result is obtained:

[2] These are the process ids assigned to the Writers by Spin.

```
(Spin Version 3.2.4 -- 10 January 1999)
      + Partial Order Reduction

Full statespace search for:
      never-claim              - (not selected)
      assertion violations     +
      cycle checks       - (disabled by -DSAFETY)
      invalid endstates  +

State-vector 24 byte, depth reached 12, errors: 0
      13 states, stored
      12 states, matched
      25 transitions (= stored+matched)
       8 atomic steps

hash conflicts: 0 (resolved)
(max size 2^19 states)
2.542  memory usage (Mbyte)
```

Verification shows that the system does not violate any safety properties. Note that the never claim is not taken into account for checking safety. Therefore, we know that SPIN has explored the entire state-space, which consists of 13 states. In the discussions that follow, we call the state-space of the system in the absence of Büchi automata the *original* state-space.

In the second phase, we check progress property *WRITER* by including its corresponding never-claim. We check the property under weak fairness, and obtain the following result:

```
State-vector 32 byte, depth reached 168, errors: 1
      22 states, stored (138 visited)
      61 states, matched
     199 transitions (= visited+matched)
      89 atomic steps
```

As expected from our experience using the LTSA, property *WRITER* is violated. Even though the on-the-fly algorithm that SPIN implements stops the graph exploration after detecting an error, the explored state-space is larger than the original one. This is due to the added never-claim. We can also see that states have been visited multiple times (22 states stored, vs. 138 states *visited*), due to the weak fairness assumption and the nested depth-first search.

SPIN additionally supports a more intuitive way of expressing progress properties. This is achieved by marking selected states in the specification as progress states. SPIN 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 way this is performed is by adding to the system a pre-defined Büchi automaton for the property.

To try the SPIN progress-checking approach for property *WRITER*, we label the states `WT` of writers as “`progressWT`”, and check the system for progress violations. The following result is then obtained:

```
Full statespace search for:
      Never-claim              +
      assertion violations     + (if within scope of claim)
      non-progress cycles      + (fairness enabled)
      invalid endstates  - (disabled by never-claim)

State-vector 32 byte, depth reached 119, errors: 1
      13 states, stored (80 visited)
      31 states, matched
     111 transitions (= visited+matched)
      51 atomic steps
```

An error is now found after 13 states. The state-space is therefore still larger than the original one, since 13 states now correspond to *part* of the state-space. However, it is smaller than the state-space obtained previously. The reason is that the pre-defined automaton used for progress properties has fewer states than the one generated automatically by the LTL translator. This confirms the fact that automatic algorithms do not normally generate the minimal automaton for LTL formulas. The following versions of the Readers/Writers problem therefore use the SPIN progress mechanism to check progress properties.

Note that 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. A conjunction of progress conditions can be dealt with by changing the progress labels in the system and repeating verification, whereas conditional progress can only be checked by using the standard LTL checking mechanism.

Version 2

To correct the liveness property violation of version 1, we make readers defer to waiting writers, and additionally use variable `readers_turn` that lets readers and writers run alternately when competition exists for the lock. The Promela model is the second one included in the appendices. By checking the model against safety properties, we find that the original state-space consists of 52 states. The results obtained when checking liveness are illustrated below:

```
State-vector 32 byte, depth reached 51, errors: 0
     102 states, stored (250 visited)
     418 states, matched
     668 transitions (= visited+matched)
     286 atomic steps
```

We can see that the original state-space has increased by almost 2 times, which corresponds to the size of the pre-defined Büchi automaton used to express progress. Additionally, we check property *READERS*, by removing the prefix *progress* from label *progressWT*, and turning label *RD* into *progressRD*. As mentioned, the SPIN progress mechanism cannot deal with the conjunction of progress properties, so they have to be checked one at a time.

In this case, the verifier detects a violation after exploring just 29 states. Note that the LTSA reported no error for this version of the Readers/Writers model. By performing a guided simulation based on the counterexample produced, we find how the error may occur; in states where the lock is free and is required by both types of processes, it is possible for the scheduler to consistently grant the lock to a writer. This happens despite the weak fairness assumption, because as soon as a writer is granted access, the reader process is no longer enabled. Therefore, as it is not *continuously enabled*, the fairness condition need not force it to execute within finite time.

This situation can also be detected with the LTSA, by giving the writers' requests and acquires high priority. However, as discussed, we consciously chose to give low priority to release actions, as we assumed that a practical scheduler would not permit situations such as the one above. We can see that our priority scheme gives us the flexibility in trying the system under different and possibly gradually increasing stress conditions.

Version 3

Our third Promela model makes the conditions under which readers and writers acquire the lock symmetrical. This version satisfies both progress properties, under the weak fairness assumption. The original state-space consists of 164 states, as opposed to a state-space of 326 states that is explored for checking progress. We have also experimented with checking the conjunction of properties *READERS* and *WRITERS*. We have thus included the never claim corresponding to property $\Box\Diamond a \wedge \Box\Diamond b$, where *a* and *b* are defined as follows:

```
#define a (Reader[0]@RD || Reader[1]@RD)
#define b (Writer[2]@WT || Writer[3]@WT)
```

The state-space then goes up to 935 states. It is therefore clear that, when using Büchi automata, it is worth checking properties one at a time, rather than checking their conjunction.

6.3 Fairness

With an LTL model checker, a system can be checked against an LTL property *f* under an LTL fairness constraint *c*, by checking the LTL property ($c \Rightarrow f$). Therefore, any LTL model checker can incorporate fairness directly. The increase in the length of the formula to be checked may result in a non-negligible increase of the size of the automaton generated for it (the size may be exponential to the length of the formula).

Alternatively, fairness constraints can be expressed directly as Büchi automata. In that case, the intersection of the system with these constraints and with the property of interest is computed, and the resulting automaton is again checked for emptiness. Note that this intersection results in a *generalised* Büchi automaton, which is defined differently than standard Büchi automata. It is not the aim of this paper to analyse this mechanism; the interested reader is referred to [8] for details. Again, each fairness automaton of size *n* introduced in the system may increase the state-space by *n* times, in the worst case.

In general, capturing fairness assumptions by means of automata may increase the state-space. Again, fairness properties express purely liveness constraints on the executions of the system, and would therefore not be expected to reduce the system state-space. Moreover, fairness assumptions are not easy to express in LTL, which is another factor that may discourage non-expert users from using tools that are based on this approach. This is particularly the case for generic fairness assumptions such as weak or strong fairness. In general, we believe that it is useful to provide the users with a pre-defined optional fairness assumption. This is taken into account both in SPIN and LTSA, as already discussed.

The difference in the fairness assumptions supported by the two tools is largely determined by the difference in analysis approaches. SPIN uses an on-the-fly approach to analysis, which preserves information about states and actions of individual processes, whereas we use CRA, where this information is not preserved under composition. Therefore, we found that the notion of fairness with respect to *transitions in the global system* fits more naturally with our framework. As mentioned, SPIN's weak fairness option theoretically increases the state-space to be explored by n+1 times, in the worst case, where *n* is the number of processes in the system. Far from introducing any overhead, fair choice is incorporated very naturally in all our liveness-checking mechanisms, whether related to progress properties, or to properties expressed as Büchi automata [6].

SPIN's weak fairness constraint is a reasonable minimal assumption in the context of most applications. However, it is often the case that additional constraints need to be imposed in order to express stronger assumptions on a system's scheduler, in which case the standard LTL approach has to be applied. Rather than starting in a conservative way, with minimal assumptions about a system's scheduler, our approach starts from constraining the scheduler with a strong assumption, and then gradually stresses the system as required. A justification for this is that we prefer to initially avoid misleading property violations, at the risk of missing problems that may occur in practice. These problems can be uncovered gradually, by imposing adverse scheduling conditions by means of action priority.

6.4 Conclusion

This section briefly summarises the issues illustrated by our case study. It is clear that Büchi automata, although very expressive, impose an overhead on verification algorithms both in terms of space and time (the former being of course a greater concern in model checking). Progress properties, although less expressive, present a clear, intuitive meaning to users, besides avoiding the above disadvantages. Moreover, several such properties can be checked during a single exploration of the state-space. When Büchi automata are used, it is recommended to check one property at a time, rather than the conjunction of these properties.

The algorithms applied in checking progress properties can operate on the fly, in order to avoid checking the entire state-space when an error is detected. Our experience with SPIN has shown that on-the-fly techniques may considerably reduce the state-space to be explored, in the presence of errors. In the context of CRA, on-the-fly mechanisms may be used at the last level of composition, where the state-space of the global system is computed based on that of its immediate sub-components.

We have also seen that, imposing action priority on a system checked under fair choice offers flexibility in expressing a variety of assumptions on the scheduler of a system. In general, it is expected that the users will use action priority to gradually increase the stress under which the system is checked. Rather than as a burden to users, we view this as adding power to the tool, with a mechanism that is easy to understand and handle.

7 Experience and Applications

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. This also coincides with the experience of others. In their work on patterns in property specifications [12], 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, for $a=true$, $\Box(a \Rightarrow \Diamond b)$ becomes $\Box\Diamond b$, which corresponds to a progress property. Additionally, checking the conditional progress property "`progress Response = if {a} then {b}`" guarantees that $\Box(a \Rightarrow \Diamond b)$ holds in all executions where *a* occurs infinitely often, i.e. in executions where $\Box\Diamond a$ holds. This is formulated in the following theorem:

Conditional Progress Theorem – The LTL formula $A = (\Box\Diamond a \Rightarrow \Box(a \Rightarrow \Diamond b))$ is equivalent to the LTL formula $B = (\Box\Diamond a \Rightarrow \Box\Diamond b)$, which can be expressed as a conditional progress property.

Proof:
If. We first prove that $A \Rightarrow B$, by proving that $\neg B \Rightarrow \neg A$.
Assume that $\neg B = \Box\Diamond a \wedge \neg \Box\Diamond b$ is true, which means that both $\Box\Diamond a$ and $\neg\Box\Diamond b$ are true. This implies that $\exists$ time instant t_i where *a* is true and $\Diamond b$ is false. Therefore, at time instant t_i, $(a \Rightarrow \Diamond b)$ is false, which implies that $\Box(a \Rightarrow \Diamond b)$ is also false. We have that:

$$\neg B \Rightarrow (\Box\Diamond a \wedge \neg\Box(a \Rightarrow \Diamond b)) \qquad (1)$$

$$\neg A = (\Box\Diamond a \wedge \neg\Box(a \Rightarrow \Diamond b)) \qquad (2)$$

From (1) and (2) we conclude that $\neg B \Rightarrow \neg A$.

Only if. We prove that $B \Rightarrow A$. It is straightforward to show that $\Box\Diamond b \Rightarrow \Box(a \Rightarrow \Diamond b)$, by the fact that $\Box(a \Rightarrow \Diamond b) = \Box(\neg a \vee \Diamond b)$. So we have that:

$$B = \Box\Diamond a \Rightarrow \Box\Diamond b \qquad (1)$$

$$\Box\Diamond b \Rightarrow \Box(a \Rightarrow \Diamond b) \qquad (2)$$

From (1) and (2) we conclude that $(\Box\Diamond a \Rightarrow \Box\Diamond b) \Rightarrow (\Box\Diamond a \Rightarrow \Box(a \Rightarrow \Diamond b))$. Therefore $B \Rightarrow A$. ■

In many cases, one wishes to check that a system satisfies a response property $\Box(a \Rightarrow \Diamond b)$, on condition that the triggering action *a* occurs regularly in the execution. For example, the user might accept the fact that a client may never receive a reply to its last request before crashing. The conditional progress theorem shows that, in those cases, it suffices to simply check the corresponding conditional progress property $\Box\Diamond a \wedge \neg \Box\Diamond b$. This property will indeed accept by default all executions where $\Box\Diamond a$ does not hold. However, on executions where it does, it will check that *b* also occurs infinitely often.

Now consider the case where a process P remains blocked waiting for b after performing a. Such situations are not detected by property ($\Box\Diamond a \Rightarrow \Box\Diamond b$). However, the user may then additionally check progress property $\Box\Diamond\alpha P$, where αP is the alphabet of process P. The latter property checks whether it is possible for a process P to remain permanently blocked after some point. For instance, let us return to the initial *PHONES* system of section 2, and substitute *PHONE(2)* with a "call_waiting" phone which, when the line is engaged, simply waits to be connected:

```
CW_PHONE(Id = 0) = (call[Id] ->
                       ( connect[Id] -> end[Id] -> CW_PHONE
                       | engaged[Id] -> WAIT)),
WAIT = (connect[Id] -> CW_PHONE).

|| CW_PHONES = ( forall [i:1..Phones-1] PHONE(i)
              || CW_PHONE(Phones) || SWITCH).
```

In this system, one can check the following combination of progress properties:

```
progress GET_THROUGH[i:1..Phones] = if {call[i]} then {connect[i]}
progress BLOCKED[i:1..Phones] = {call[i],connect[i],end[i],engaged[i]}
```

The following violation is then reported:

```
Progress violation: BLOCKED.2
Trace to terminal set of states:
      call.1
      connect.1
      call.2
      engaged.2
Actions in terminal set:
{call.1, connect.1, end.1}
```

We can see that, although the conditional progress property *GET_THROUGH* is satisfied, *PHONE(2)* may block as a result of an incompatibility between its call-waiting behaviour, and the behaviour of the switch. We believe that in general, by combining simple and conditional properties, developers can avoid the use of the general response pattern, in most cases of interest.

In some circumstances, a response pattern $\Box(a \Rightarrow \Diamond b)$ may need to be made stricter by requiring that exactly one b action corresponds to each a action. For example, a client server system may need to ensure that exactly one reply is produced for each client request. This is a combination of a progress and a safety property. The progress property simply expresses that $\Box(\mathit{request} \Rightarrow \Diamond \mathit{reply})$. The safety property requires that requests and replies alternate themselves, without forcing the actual occurrence of either of these actions. Our approach to modelling and analysing safety properties is described in [5]. In general, our analysis methodology separates the phases of safety and liveness analysis. This is due to the fact that specialised and efficient algorithms have been developed for each type of property. However, it also reflects the different concerns of the user at each one of these phases [6]. For example, it is critical to first ensure that a system never enters an unsafe state, before finding out whether specific good events eventually happen. Note that any property that can be expressed as a Büchi automaton can be separated algorithmically into a pure liveness and a pure safety property [22].

We have applied our approach to a number of case studies. For example, on a model of an Active Badge System with 566,820 states and 2,428,488 transitions [2], we showed that badge commands are not acknowledged if badges move between locations too frequently.

The combination of progress checks and action priority also 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 [23]. Then for a closed system, the maximal progress condition that is usually assumed for discrete time models is ensured by making the tick action low priority: “`>> {tick}`”. We consider a system as *closed*, when the interesting part of the behaviour of its environment has been included in the model, and therefore the system is not expected to interact with components that are not part of it. In this case, all actions in the system can be considered as internal, and therefore they are *urgent* with respect to the passage of time. Additionally, the fact that a system is free of both zeno behaviours, and of time deadlocks, can be demonstrated by asserting the progress property: “`progress NoZeno_NoDeadlock = {tick}`”.

We have used these principles in a number of small case studies [13]. We have also used them to construct and check a discrete time model for a Bounded Retransmission Protocol used in one of Philips’ products. This is a standard industrial case study used to demonstrate the strength of verification techniques in handling real-time systems. We have managed to prove the correctness of this protocol with respect to both safety, and liveness properties. Moreover, our technique has helped us to establish the minimum timeout values that guarantee correctness of the protocol. Our experience is described in detail in [24].

8 Related Work

Progress. Manna and Pnueli classify properties of programs into a hierarchy, where each class is characterised by a canonical temporal formula scheme [18]. 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 [18] as *response*. The notion of progress also appears in Unity [25], 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 $a\boldsymbol{U}b$ (ensures), where $\boldsymbol{U}$ denotes strong until.

As mentioned, SPIN [26] uses the notion of progress in a similar context to ours. Our approach differs significantly from that of SPIN both in terms of expressiveness, and algorithmically, as discussed in section 6.

Fairness. The issue of fairness has been extensively investigated. Lehmann *et al.* introduced three notions of fairness that are useful in practice [27]. 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 [28] or actions [29]. 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 transitions in the global system. Different notions of fairness are appropriate for different system models. Apt et al. [30] present some criteria of effectiveness and utility of adopting some notion of fairness in a computational model.

Queille and Sifakis [16] stress the importance of defining fairness with respect to specific actions or predicates of the system, which they call relative fairness. Natarajan and Cleaveland [31] 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 [18] supports the specification of weak and strong fairness with respect to specific system transitions.

An interesting approach to fairness in a compositional setting is the one associated with I/O automata [32]. An I/O automaton definition includes a “task partition”, i.e. a partition on its locally controlled actions. The fairness assumption made on such automata is then weak fairness with respect to each equivalence class in the task partition. In other words, fairness states that, for each equivalence class *C*, a fair infinite execution of an I/O automaton contains infinitely many actions from *C*, or infinitely many states in which no action in *C* is enabled. Weak fairness with respect to the component processes of a system can then be expressed by having the alphabet of each process constitute an equivalence class in the task partition of the system I/O automaton.

A way of dealing with fairness in model checking is to add Büchi acceptance conditions to the system. For example in [9], 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 [8] describe how a Büchi automaton can be used to express a fair process scheduler. The branching logic CTL cannot express the fact that properties are required to hold only along fair *executions*. For this reason in [28], Clarke *et al* extend their model with a set of state predicates, and define fair paths as paths where each one of these predicates holds infinitely often. This is equivalent to turning the model of the system into a generalised Büchi automaton. In this framework, they describe how to express weak and unconditional fairness with respect to processes. Their proposed mechanism requires the user to modify the initial model of the system. More recent work by Clarke *et al* [33] reports that fairness constraints are expressed as CTL formulas that must hold infinitely often along any fair path. Their symbolic model-checking algorithms take fairness into account. Finally, in Unity [25], 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 [34], 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 [35]. 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 [36].

9 Discussion and Conclusion

The work presented in this paper was motivated by a desire to achieve a balance between expressive power, accessibility and efficiency of analysis methods. As demonstrated in this paper, 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 [20]. 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 [37]. 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 specific classes of liveness properties. Our experience and that of others 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. Additionally, the combination of progress checks and action priority provides an elegant way of dealing with models that incorporate a notion of discrete time.

In the context of liveness property checking, the possibility of including a notion of fairness is essential. Moreover, as argued in earlier sections, it is important for a tool to provide a predefined optional fairness assumption, that could be easily applied by users who don't have the expertise to express such assumptions explicitly. We have taken a non-conservative approach to the issue, which imposes a strong constraint on the scheduler of a system. This simplifies the liveness-checking mechanisms, and avoids imposing any additional time or space overheads to them. In general, we believe that, rather than checking liveness with no or very weak 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 priorities 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.

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. In 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, Gerard Holzmann and Dragan Boshnachki for answering SPIN-related questions, and Martti Tienari for comments related to our concept of fair choice.

References

[1] 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: pp. 7-35.

[2] 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*. October 1997, Tunis, Tunisia, pp. 240-245.

[3] 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)*. April 16-18 1998, Ise-shima, Japan, pp. 144-146.

[4] 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)*. September 1997, Zurich, Switzerland. Springer, Lecture Notes in Computer Science 1301, pp. 227-243. M. Jazayeri and H. Schauer, Eds.

[5] Cheung, S.C. and Kramer, J., *Checking Safety Properties Using Compositional Reachability Analysis.* ACM Transactions on Software Engineering and Methodology, Vol. **8**(1), January 1999: pp. 49-78.

[6] Giannakopoulou, D., “Model Checking for Concurrent Software Architectures”, PhD Thesis, March 1999.

[7] 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*. June 1986, Cambridge. IEEE Computer Society Press, pp. 322-331.

[8] Gribomont, P. and Wolper, P., "Temporal Logic", in *From Modal Logic to Deductive Databases*, A. Thayse, Editor, 1989, John Wiley and Sons.

[9] 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), April 1990: pp. 303-339.

[10] 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)*. June 1993, Liège, Belgium. North-Holland, pp. 109-124. A. Danthine, G. Leduc, and P. Wolper, Eds.

[11] 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)*. June 1995, Warsaw, Poland, pp. 3-18.

[12] 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)*. 16-22 May 1999, Los Angeles, CA. ACM, pp. 411-420.

[13] Magee, J. and Kramer, J., *Concurrency: State Models & Java Programs*: John Wiley & Sons, 1999.

[14] Hoare, C.A.R., *Communicating Sequential Processes*: Prentice-Hall, 1985.

[15] Tarjan, R., *Depth-First Search and Linear Graph Algorithms.* SIAM Journal of Computing, Vol. **1**, 1972: pp. 146-160.

[16] 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.

[17] Andrews, G.R., *Concurrent Programming - Principles and Practice*: The Benjamin / Cummings Publishing Company Ltd., 1991.

[18] Manna, Z. and Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems - Specification*: Springer-Verlag, 1992.

[19] Aho, A.V., Hopcroft, J.E., and J.D.Ullman, *The Design and Analysis of Computer Algorithms*: Addison-Wesley.

[20] Holzmann, G.J., *The Model Checker SPIN.* IEEE Transactions on Software Engineering, Vol. **23**(5), May 1997: pp. 279-295.

[21] Choueka, Y., *Theories of automata on omega-tapes: a simplified approach.* Journal of Computer and System Sciences, Vol. **8**, 1974: pp. 117-141.

[22] Alpern, B. and Schneider, F.B., *Recognising safety and liveness.* Distributed Computing, Vol. **2**: pp. 117-126.

[23] Roscoe, A.W., *The Theory and Practice of Concurrency*: Prentice Hall, 1998.

[24] Giannakopoulou, D., "Modelling and Analysis of the Bounded Retransmission Protocol: Experience with Discrete Time", Dept. of Computing, Imperial College, London, Research Report (Under Preparation), 1999.

[25] Chandy, K.M. and Misra, J., *Parallel Program Design: a Foundation*: Addison-Wesley, 1988.

[26] Holzmann, G.J., *Design and Validation of Computer Protocols*. Prentice Hall Software Series: Prentice Hall, 1991.

[27] 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*. July 13- 17, 1981, Acre (Akko), Israel. Springer, Berlin, Lecture Notes in Computer Science 115, pp. 264-277. S. Even and O. Kariv, Eds.

[28] 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), 1986: pp. 244-263.

[29] Lamport, L., *The Temporal Logic of Actions.* ACM Transactions on Programming Languages and Systems, Vol. **16**(3), May 1994: pp. 872-923.

[30] Apt, K.R., Francez, N., and Katz, S., *Appraising fairness in languages for distributed programming.* Distributed Computing, Vol. **2**, 1988: pp. 226-241.

[31] Natarajan, V. and Cleaveland, R. "Divergence and Fair Testing", in *Proc. of the Automata, Languages and Programming (ICALP '95).* July 1995, Szeged, Hungary. Springer-Verlag, Lecture Notes in Computer Science 944, pp. 648-659. Z. Fulop and F. Gecseg, Eds.

[32] Lynch, N.A., *Distributed Algorithms*: Morgan Kaufmann Publishers, Inc., 1996.

[33] Clarke, E.M., Grumberg, O., and Long, D., "Model Checking", in *Springer-Verlag Springer-Verlag Nato ASI series F,* Vol. 152, 1996.

[34] Phillips, I., "Approaches to priority in process algebra", Dept. of Computing, Imperial College, London, Draft Report, 1994.

[35] 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).* November 1997, Osaka. Chapman and Hall, pp. 321-336. T. Mizuno, *et al.*, Eds.

[36] Cleaveland, R. and Hennessy, M., *Priorities in process algebra.* Information and Computation, Vol. **87**(1/2), July/August 1990: pp. 58-77.

[37] 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.

APPENDIX

```
/* first SPIN program: plain Readers/Writers */

#define R 2
#define W 2
#define a  (Writer[2]@WT || Writer[3]@WT)

byte readers = 0;
bool writing = false;


active [R] proctype Reader ( ) {
             do  ::
             atomic {       /* acquire read */
               (!writing && readers<=R) -> readers++;
             }
RD:          assert (!writing);  /* now reading */
             readers--;  /* release read */
             od
}

active [W] proctype Writer ( ) {
             do  ::
             atomic {       /* acquire write */
               (!writing && readers==0) -> writing = true;
             }
WT:          assert (readers==0); /* now writing */
             writing = false;  /* release write */
             od
}

never {    /*  !([]  <>  a)  */
T0_init:
             if
             :: (! ((a))) -> goto accept_S2
             :: (1) -> goto T0_init
             fi;
accept_S2:
             if
             :: (! ((a))) -> goto T0_S2
             fi;
T0_S2:
             if
             :: (! ((a))) -> goto accept_S2
             fi;
accept_all:
             skip
}
```

```
/* second SPIN program: using readers_turn variable, version 1 */

#define R 2
#define W 2

byte readers = 0;
bool writing = false;
byte wwaiting = 0;
bool readers_turn = true;


active [R] proctype Reader ( ) {
             do  ::
             atomic {       /* acquire read */
              ((!writing && readers<=R) &&
                        (wwaiting==0 || readers_turn))
               -> readers++; readers_turn = false;
             }
RD:          assert (!writing);
             readers--;  /* release read */
             od
}

active [W] proctype Writer ( ) {
                 do  ::
                 wwaiting++;
                 atomic {       /* acquire write */
                   (!writing && readers==0)
                     -> writing = true; readers_turn = true;
                 }
progressWT:      wwaiting--;   /* writing */
                 assert (readers==0);
                 writing = false;  /* release write */
                 od
}

/* In order to check property READER, label RD is turned into label
progressRD, and label progressWT is turned into WT */
```

```
/* third SPIN program: using readers_turn variable, version 2 */

#define R 2
#define W 2

byte readers = 0;
bool writing = false;
byte rwaiting = 0;
byte wwaiting = 0;
bool readers_turn = true;


active [R] proctype Reader ( ) {
             do  ::
             rwaiting++;
             atomic {       /* acquire read */
               ((!writing && readers<=R) &&
                        (wwaiting==0 || readers_turn))
                -> readers++; readers_turn = false;
             }
RD:          rwaiting--;   /* reading */
             assert (!writing);
             readers--;  /* release read */
             od
}

active [W] proctype Writer ( ) {
             do  ::
             wwaiting++;
             atomic {       /* acquire write */
               ((!writing && readers==0) &&
                        (rwaiting==0 || !readers_turn))
                -> writing = true; readers_turn = true;
             }
WT:          wwaiting--;   /* writing */
             assert (readers==0);
             writing = false;  /* release write */
             od
}
```