Progress Checking for Dummies

Antti Valmari[1] and Henri Hansen[2]

[1] University of Jyväskylä, FINLAND
antti.valmari@jyu.fi
[2] Tampere University of Technology, Mathematics
P.O. Box 553, FI–33101 Tampere, FINLAND
henri.hansen@tut.fi

Abstract. Verification of progress properties is both conceptually and technically significantly more difficult than verification of safety and deadlock properties. In this study we focus on the conceptual side. We make a simple modification to a well-known model to demonstrate that it passes progress verification although the resulting model is intuitively badly incorrect. Then we point out that the error can be caught easily by adding a termination branch to the system. We compare the use of termination branches to the established method of addressing the same need, that is, weak fairness. Then we discuss another problem that may cause failure of catching progress errors even with weak fairness. Finally we point out an alternative notion of progress that needs no explicit fairness assumptions. Our ideas are especially well-suited for newcomers in model checking, and work well with stubborn set methods.

Keywords: usability of verification methods; progress; fairness; fair testing

1 Introduction

To motivate the present study, let us consider the example system in Fig. 1. It shows Peterson's famous mutual exclusion algorithm [5], and this particular model appears on the home page spinroot.com of the SPIN verification tool.

SPIN reports no errors in this model, as expect, because Peterson's algorithm is correct under the usual assumptions on the execution model. Swapping lines 7 and 8 and running SPIN results in `assertion violated (ncrit==1)`. This is because in the modified system the following scenario is possible: First process 0, followed by process 1, executes `turn = _pid`. Now `turn` = 1. Next process 1 continues to the critical section. It passes line 9 because process 0 has not yet executed `flag[_pid] = 1`, due to swapping lines 7 and 8. Finally process 0 continues to the critical section. It passes line 9 because `turn` = 1. As both processes are now in the critical section, the assertion is violated on line 11. We call the swapping of lines 7 and 8 *modification A* from now on.

Let us now take the original model and remove `|| turn == 1 - _pid` on line 9, and call this *modification B*. For this model, SPIN reports `invalid end state`, because when both processes execute lines 7 and 8, both flags contain

```
1  bool turn, flag[2];         // the shared variables, booleans
2  byte ncrit;                 // nr of procs in critical section
3  active [2] proctype user()  // two processes
4  {
5      assert(_pid == 0 || _pid == 1);
6  again:
7      flag[_pid] = 1;
8      turn = _pid;
9      (flag[1 - _pid] == 0 || turn == 1 - _pid);
10     ncrit++;
11     assert(ncrit == 1); // critical section
12     ncrit--;
13     flag[_pid] = 0;
14     goto again
15 }
```

Fig. 1. Peterson's algorithm from the SPIN homepage

the value 1, and neither process can pass the modified line 9. That is, the system is in a deadlock.

We have seeded two different errors to the model, and SPIN has caught them immediately. Finally, let us take the original model and remove `flag[1 - _pid] == 0 ||` from line 9. Let us call this *modification C*. This time SPIN reports no errors.

Should we conclude then, that `flag[1 - _pid] == 0 ||` is unnecessary in Peterson's algorithm? In its absence, if a process wants to enter the critical section, it has to wait at line 9 until the other process also seeks entry. Then the former process can enter the critical section. The latter process has to wait until the former process has left the critical section, which is acceptable behaviour. However, the latter process has to wait further still: The latter process is prevented from entering the critical section as long as the former process is not there and does not want to go there; it can only enter *after* the former process has requested entry. Intuitively, this is clearly unacceptable.

The problem is that the (implicit) correctness specification in Fig. 1 is insufficient, and it fails to declare the above scenario as illegal.

Let us now make a small modification to Fig. 1. The modified model is shown in Fig. 2. The cycle consisting of `again:` and `goto again` has been replaced by a `do`–`od`-cycle, and a line has been added that makes it possible for each process to exit the cycle. Immediately after exiting the process terminates. Each time when on line 6, the process chooses nondeterministically between terminating and trying to execute the statements in Fig. 1. SPIN reports no errors in this model, and if modification A or B is made to this model, SPIN gives the same error reports as before.

However, now also modification C causes SPIN to report an error. It reports `invalid end state`. Indeed, if one process terminates and the other goes to line 9, the system is in a deadlock, because the process on line 9 cannot continue.

```
1   bool turn, flag[2];         // the shared variables, booleans
2   byte ncrit;                 // nr of procs in critical section
3   active [2] proctype user()  // two processes
4   {
5       assert(_pid == 0 || _pid == 1);
6   do
    ::  break
    ::
7       flag[_pid] = 1;
8       turn = _pid;
9       (flag[1 - _pid] == 0 || turn == 1 - _pid);
10      ncrit++;
11      assert(ncrit == 1); // critical section
12      ncrit--;
13      flag[_pid] = 0;
14  od
15  }
```

Fig. 2. Peterson's algorithm with a termination branch

The theme of the present study is how so-called *progress errors* can sometimes be caught with small tricks that are easier than the standard approach and less vulnerable to accidental misuse that causes failure of catching errors. The addition of the termination branch as was done in Fig. 2 is such a trick. The standard method of obtaining the same effect is via so-called weak fairness assumptions [3], but it is so much more difficult that it was not done in the original model. We do not claim that our tricks cover all progress properties, only that they are an *easy-to-use alternative* that is much better than finding the standard method too difficult to use and therefore not trying to verify progress properties at all. Our tricks thus address usability shortcomings of existing methods with respect to their industrial applicability.

Furthermore, we do not claim that our observations and tricks are fundamentally new. As a matter of fact, the key ideas are two decades old [2, 10]. We do claim, however, that their benefits are not sufficiently widely known or have been under-appreciated. Although the addition of the termination branch is easy, gives tangible added value, and is almost free from drawbacks (it makes the size of the state space grow a bit), it seems that it is seldom done. The modelling style in Fig. 1 seems to be the norm, for example among the BEEM benchmarks [4], most models of clients in scheduler or resource allocation systems lack a similar voluntary termination branch.

In Section 2 we analyse why the model in Fig. 1 failed and the model in Fig. 2 succeeded in revealing the error. We also compare the addition of the termination branch with the standard solution to the same problem using weak fairness. In Section 3 we discuss a theory that makes it possible to design a model of the user of a service that assumes as much as necessary, but not more, about the behaviour of the user. There is also another kind of problem that may cause an

intuitively incorrect model to accidentally pass model checking. It will be solved in Section 4. An easy generic approach to progress that is slightly weaker than the standard approach is discussed in Section 5.

Full formal treatment of the material in this study would require repeating numerous definitions that can be found in the literature. Because of lack of space, we focus on the intuition and present formally only the concepts which, we believe, are neither obvious nor widely known. Please note that even though the proposed method is suitable "for dummies", this article may not be.

2 Unforced Request

In this section we discuss why and how the addition of the termination branch facilitates the detection of the error caused by modification C. We recall an analogous solution in linear temporal logic [3]. Lastly, we discuss the error detection power and technical difficulty of the methods. When we refer to Fig. 1 or 2, unless otherwise mentioned, we mean both the original figure and modifications A, B and C.

We must first distinguish between a system and its correctness requirements. Lines 2, 10, 11 and 12 of Fig. 1 are not part of Peterson's algorithm, for example, [5] contains nothing corresponding to them. Instead, they express a correctness requirement. They specify that the system should ensure that the two processes are never in the critical section at the same time. This property is usually called *mutual exclusion*. If line 11 is removed, the model loses its ability to catch the error caused by modification A.

There is also an *implicit* correctness requirement arising from the semantics of Promela and the default behaviour of the SPIN tool: in all terminal states, both processes must be at the end of their code, that is, on line 15. (Also line 5 is not part of the algorithm. It expresses a low-level technical correctness requirement related to debugging Promela specifications, and is not important for our discussion.)

In addition to safety properties, systems are usually required to satisfy some progress properties, which in mutual exclusion or resource allocation systems would be called *eventual access*. It says that if a process has requested for access to a resource, then it will eventually get it. In the case of Fig. 1 and 2, a process requests for access by assigning 1 to its flag, that is, by executing line 7. Therefore, eventual access is violated if and only if a process reaches line 8 but then fails to reach line 11. Let P_i denote the line where process i is (at the beginning of the line). In the case of Fig. 1 and 2, eventual access can be specified in linear temporal logic as $\Box(P_0 = 8 \rightarrow \Diamond(P_0 = 11))$ and $\Box(P_1 = 8 \rightarrow \Diamond(P_1 = 11))$.

Figures 1 and 2 specify eventual access, to the extent they specify it, implicitly, via the requirement that the model must not stop while a process is on line 8, 9, or 10 (which follows from the above-mentioned requirement that the model must not stop while a process is on any other line than 15). Indeed, the error caused by modification B is a violation of eventual access, because when both processes are waiting on line 9, process 0 has requested for access but will

never get to line 11 (and the same holds also on process 1). In Section 1, the error was caught as an unintended terminal state. The users of linear temporal logic would catch it as a violation of eventual access. We will return to this difference towards the end of this section. For the time being, we focus on eventual access.

In the case of Fig. 2, but not Fig. 1, also the error caused by modification C is a violation of eventual access. In the case of Fig. 1 with modification C, eventual access holds formally, and thus it is formally correct not to report any error although intuitively the system is badly wrong. The failure of the model in Fig. 1 to catch the error caused by modification C is *not* due to insufficient specification of correctness requirements, but due to implicit *over-specification* of the users of the algorithm. This is a subtle issue that we will discuss next.

It is obvious that if one process stays in the critical section forever and the other process requests for access, then either safety or progress is violated: If the algorithm lets also the other process enter the critical section, then mutual exclusion is violated, and if it does not, eventual access is violated. This means that to solve the mutual exclusion problem, it is necessary to assume *something* about the behaviour of the clients. In particular, it is necessary to assume that a client will not stay in the critical section forever.

Like most modelling languages and model checking tools, Promela and SPIN implicitly assume that *if something can happen* in the model, *then something will happen*. Most of the time this is a very appropriate assumption. Among other things, if a process of Fig. 1 or 2 is in the critical section, the assumption forces it to leave it at the latest when the other process is waiting on line 9 or 15.

On the other hand, in the case of Fig. 1 with modification C, this implies that each process will always eventually request for access. This is because if we try to execute the model so that one process stays on line 7, eventually the other process reaches line 9, and the only thing that can happen in the model is that the former process executes line 7.

Real-life users of such systems do not necessarily always eventually request for access. This means that Fig. 1 makes an unjustified assumption about the user. It is this assumption that makes Fig. 1 with modification C pass formal verification, although it is intuitively badly incorrect. To avoid this problem, a model where requests are issued should exhibit what we call *unforced request*. Informally, it says that each process must be able to choose not to request for access. We do not call it a property, to emphasize that it is not something that the model checker should check about the system. Instead, it should be enforced by building the model appropriately. It is not a restriction on the behaviour of the processes; it is a requirement that a certain restriction is *not* made.

We conclude that there are two distinct reasons why model checking may fail to reveal an error: *under-specification of the requirements* (such as leaving out line 11) and *over-specification of the model* (such as assuming that each process will always eventually request for access). In Section 3 we will discuss over-specification of the model in a solid theoretical framework that makes the notion formally precise. In Section 4 we will encounter a third reason. We continue this section by comparing two methods of implementing unforced request.

Figure 2 implements unforced request by adding a termination branch. When a process is on line 6, it is not forced to go to and execute line 7 even when the other process is waiting on line 9, because it can nondeterministically choose to execute the `break` statement instead, thereby going to line 15.

In order to present a theorem later in this section, we make it more formal what we mean by systems, processes, and termination branches. Due to lack of space, we only discuss a minimal set of concepts that will be needed in the sequel. A *system* consists of processes and variables. A *process* is a rooted directed edge-labelled graph whose vertices are called *(local) states*, edges are called *transitions*, and the root is called *initial state*. In addition to the tail and head states, a transition has a *guard* and a *body*. The guard is a Boolean function on the values of the variables. The body assigns values to zero or more variables as a function of the values of the variables.

For instance, in the case of Promela, the set of states is implicit from the code. Line 8 of Fig. 1 expresses a transition whose guard is identically `true` and whose body may change the value of `turn`. Line 9 expresses a transition whose tail state is the head state of the transition on line 8, body makes no assignments, and guard is the condition written on the line.

The addition of a *termination branch* to state s of a process means the addition of one state s' and one transition whose tail state is s, head state is s', guard is the constant function `true`, and body makes no assignments.

In linear temporal logic, instead of adding a termination branch, it is customary to use so-called *weak fairness* assumptions. Intuitively, weak fairness towards transition t means the assumption that if t is enabled for long enough, it will eventually be executed. An execution is thus *weakly unfair* towards t if and only if, from some point on, t is enabled in every state but does not occur. Weakly unfair executions are not treated as valid counterexamples to a property. Typically weak fairness is assumed towards almost all transitions. Not assuming weak fairness towards a transition is thus exceptional and indicates that the transition need not occur even if it is enabled. Weak fairness may also be assumed towards a set of transitions, but we skip that.

The assumption can be thought to reflect that processor time allocation of a real system works well enough that weakly unfair executions do not occur. Strictly speaking, only infinite executions can be weakly unfair, but by a weakly unfair execution in the real world, we mean an execution where t is enabled for "too long" without occurring. Real schedulers are not guaranteed to work that well, of which the Mars Pathfinder priority inversion incident of 1997 [6] is an example (to the extent that such an example can exist). However, to avoid problems like this, schedulers are usually designed to guarantee (some real-world approximation of) weak fairness. This is why it is usually considered reasonable to assume weak fairness in model checking with Promela-like languages. As was discussed in [1], weak fairness may not work as well with process algebras.

To indicate that a process need not execute line 7 if it does not want to, weak fairness is assumed towards every transition except the one that corresponds to

line 7. This implies that line 7 need not be executed, even if it is the only thing that can happen in the model.

This method has the advantage that violations of eventual access are caught independently of whether the problem is a deadlock or something else. For instance, assume that line 9 of Fig. 1 is replaced by

```
if
::  flag[1-_pid] == 1 -> flag[_pid] = 0; goto again
::  else -> skip
fi;
```

Let us call this *modification D*. A scenario becomes possible where both processes repeatedly go to this modified line 9, detect that also the other process has made the request, cancel their own request, and go back. This cycle can repeat forever, resulting in neither process ever getting to the critical section. It is weakly fair towards every transition, and thus valid as a counter-example to eventual access. In this case SPIN does not detect the error, but it could be made to detect the error by using standard techniques for linear temporal logic that rely on detecting certain kinds of cycles in the state space.

On the other hand, this approach is more complicated both for the modeller and for the verification tools. Indeed, the authors of spinroot.com were wise enough not to use linear temporal logic in the example, to keep it simple enough to act as a first example to a newcomer who is not familiar with temporal logic. We conclude that it makes sense to have deadlock detection in the toolbox, although its ability to detect errors is restricted. Indeed, SPIN has it. Our contribution here is the remark that it can be made to detect *more* errors by adding termination branches. Although termination branches cannot catch the error caused by modification D, they did catch the error caused by modification C, which is better than nothing and took little extra effort.

We say that an execution is *complete* if and only if it is either infinite or ends in a terminal state. In linear temporal logic, it is customary to extend every complete finite execution to an infinite one by repeating its last state forever, because doing so eliminates a special case and thus simplifies the theory. Deadlocks become infinite executions where, from some point on, nothing useful happens. The notion of weak fairness does not depend on this convention, so we ignore it in the sequel. If the convention is obeyed also in the model checking tool, then no errors are caught as unexpected deadlocks.

We still have to justify that termination branches do not cause false alarms. We first need to introduce yet another concept.

Almost every linear temporal logic property that is relevant for model checking in practice is *stuttering-insensitive*. The formal definition is not important for the present study, so we skip it. Intuitively, a transition is *visible* with respect to a linear temporal logic formula if and only if its occurrence may affect the truth value of an atomic proposition in the formula. Stuttering-insensitivity means that the number of invisible transitions that occur before the first, after the last, or between any two visible transitions is irrelevant. In particular, eventual access is stuttering-insensitive, and the `break` transition in Fig. 2 is invisible

with respect to it. By $\mathrm{LTL}_{\mathsf{X}}$ we mean the set of the stuttering-insensitive linear temporal logic formulae.

The following theorem implies that violations against an $\mathrm{LTL}_{\mathsf{X}}$ formula detected with the termination branch method are errors also when using the weak fairness assumptions, assuming that the formula obeys a mild assumption. The assumption rules out such formulae as $\Box(P_1 \neq 15)$ that directly tests the presence of the termination branch that was added in Fig. 2. Due to lack of space, we formulate and prove the theorem for only a single addition of a termination branch, but the proof can be generalized to multiple additions.

Theorem 1. *Let S be a system, φ an $\mathrm{LTL}_{\mathsf{X}}$ formula on it, s a state of a process of S, and $t_1, \ldots, t_n$ be the transitions whose tail state is s, such that weak fairness is not assumed with respect to any of $t_1, \ldots, t_n$. Let S' be obtained from S by adding the termination branch $s \xrightarrow{-t'} s'$. We also assume that t' is invisible with respect to φ. If S' has a deadlocking execution that violates φ and contains t', then S has a weakly fair execution that violates φ.*

Proof. Let ξ' be the execution of S' in the claim. Because the guard of t' tests nothing and the body of t' only changes the local state of the process from s to s', the execution of t' can be removed from ξ', and the result ξ is an execution of both S and S'. It ends in s. It is weakly fair in S, because weak fairness does not require the execution of any of $t_1, \ldots, t_n$, and no other transition is enabled because ξ' is deadlocking. Because φ is stuttering-insensitive and t' is invisible with respect to it, φ has the same truth value on ξ as on ξ'. □

Please notice that the theorem holds independently of what weak fairness assumptions are made in S', if any. This is because it follows from the definition of weak fairness that all deadlocking executions are weakly fair.

3 Most General Client

In this section we discuss a theory that makes it possible to avoid over-specification of the kind discussed in the previous section. The theory will lead to the conclusion that the model used in the previous section is, in a rigorous sense, optimal for the verification of eventual access.

If S is a system and φ is an $\mathrm{LTL}_{\mathsf{X}}$-formula, then $S \models \varphi$ means that φ holds on S. The following theorem is from [9] and an earlier version appeared in [2]. It establishes a useful link between $\mathrm{LTL}_{\mathsf{X}}$ and process algebras. We will introduce the necessary process-algebraic concepts and discuss the link after the theorem.

Theorem 2. *The CFFD-semantics preserves $\mathrm{LTL}_{\mathsf{X}}$ in the following sense: If φ is an $\mathrm{LTL}_{\mathsf{X}}$-formula whose atomic propositions do not refer to the local states of Q, and if $P_1 \parallel \cdots \parallel P_n \parallel Q \models \varphi$ and $Q' \preceq_{\mathsf{CFFD}} Q$, then $P_1 \parallel \cdots \parallel P_n \parallel Q' \models \varphi$.*

In the theorem, a system is expressed as a parallel composition of *labelled transition systems*, abbreviated *LTS*. Intuitively, an LTS is the representation of the behaviour of a system, subsystem, or individual component of the system

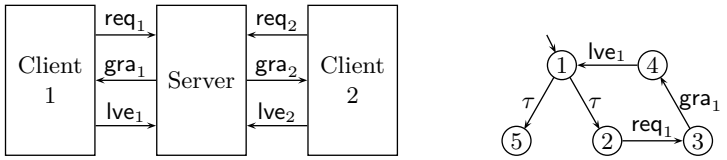

Fig. 3. (Left) A client-server mutual exclusion system (Right) Client 1 as an LTS

as an edge-labelled directed graph. It is the state space of the subsystem, with emphasis on transitions instead of states. Formally, it is a tuple $L = (S, \Sigma, \Delta, \hat{s})$. The set Σ is the set of the visible actions of L, also known as the *alphabet* of L. The symbol τ is used to denote invisible actions, and $\tau \notin \Sigma$. The set of the states of L is S, and $\hat{s} \in S$ is the initial state. The set of the transitions of L is $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$. The *reachable part* of an LTS $(S, \Sigma, \Delta, \hat{s})$ is the LTS $(S', \Sigma, \Delta', \hat{s})$, where S' and Δ' are the smallest subsets of S and Δ such that $\hat{s} \in S'$ and, if $s \in S'$ and $(s, a, s') \in \Delta$, then $s' \in S'$ and $(s, a, s') \in \Delta'$.

For building a system from its components, many different operators have been defined. We will only need the *parallel composition* operator $\|$. Intuitively, $L_1 \parallel L_2$ represents the parallel execution of L_1 and L_2, with those actions executed jointly that are in the intersection of their alphabets. Formally, it is the reachable part of $(S, \Sigma, \Delta, \hat{s})$, where $S = S_1 \times S_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\hat{s} = (\hat{s}_1, \hat{s}_2)$, and Δ is defined as follows: $((s_1, s_2), a, (s_1', s_2')) \in \Delta$ if and only if either $a \in \Sigma_1 \cap \Sigma_2$, $(s_1, a, s_1') \in \Delta_1$, and $(s_2, a, s_2') \in \Delta_2$; $a \notin \Sigma_2$, $(s_1, a, s_1') \in \Delta_1$, and $s_2' = s_2 \in S_2$; or $a \notin \Sigma_1$, $(s_2, a, s_2') \in \Delta_2$, and $s_1' = s_1 \in S_1$. Because τ is not in any alphabet, invisible actions are not executed jointly. The parallel composition operator is associative and commutative (up to isomorphism on the names of states). So $P_1 \parallel \cdots \parallel P_n \parallel Q$ is now well-defined.

Figure 3 shows the overall structure and one client of a mutual exclusion system as a client–server system. Assume that we want to verify that eventual access holds on Client 1. Then we can let $n = 1$, P_1 be Client 1 and Q be the parallel composition of Server and Client 2 in Theorem 2. Although Q is a single LTS in the theorem, we can use Server $\parallel$ Client 2 in its place, because Client 1 $\parallel$ Server $\parallel$ Client 2 = Client 1 $\parallel$ (Server $\parallel$ Client 2) by the associativity of $\parallel$.

Many process algebras are *compositional*. That is, a subsystem can be replaced by a smaller, semantically equivalent subsystem before model checking. This is a powerful tool for alleviating the state explosion problem. What is more, many process algebras have a *precongruence*. It is defined with respect to a set of operators for building systems from LTSs and subsystems. It is a partial order relation $\preceq$ between LTSs such that if $L_1 \preceq L_2$ and f is a process-algebraic expression that only uses operators from the set, then $f(L_1) \preceq f(L_2)$.

By $\preceq_{\mathsf{CFFD}}$ wean mean the preorder in the *Chaos-free Failures Divergences Semantics* [11]. It has some variants depending on the chosen set of operators for building systems. The following is appropriate for the present context.

Let Σ_i denote the alphabet of L_i. A *trace* of L_i is any element of Σ_i^* that can be obtained by picking a finite (not necessarily complete) execution of L_i and

dropping all the states and τ-symbols from it. If the same is done to an infinite execution, the resulting sequence of elements of Σ_i is either finite or infinite. If it is finite, it is called a *divergence trace*, and otherwise an *infinite trace*. The sets of divergence and infinite traces of L_i are denoted with $Div(L_i)$ and $Inf(L_i)$.

A *stable failure* of L_i is a pair $(\sigma, A) \in \Sigma_i^* \times 2^{\Sigma_i}$ such that L_i can reach a state s via an execution whose trace is σ, such that no output transition of s is labelled with any element of $A \cup \{\tau\}$. The set of the stable failures is denoted by $Sf(L_i)$. This notion generalizes traces that lead to terminal states. In particular, σ is a trace that leads to a terminal state if and only if $(\sigma, \Sigma_i) \in Sf(L_i)$. In [8] it was proven that to obtain the congruence property with respect to $\|$, the semantics must preserve all stable failures.

We define $L_1 \preceq_{\mathsf{CFFD}} L_2$ if and only if $\Sigma_1 = \Sigma_2$, $Sf(L_1) \subseteq Sf(L_2)$, $Div(L_1) \subseteq Div(L_2)$, and $Inf(L_1) \subseteq Inf(L_2)$. We also define $L_1 \approx_{\mathsf{CFFD}} L_2$ if and only if $L_1 \preceq_{\mathsf{CFFD}} L_2$ and $L_2 \preceq_{\mathsf{CFFD}} L_1$.

Intuitively, Theorem 2 says that, when $L \preceq_{\mathsf{CFFD}} L'$, replacing L by L' as a component of a system may introduce new violations of a given LTL$_{\mathsf{X}}$ property. Conversely, if a system with L' is correct with respect to a given formula, replacing it with L is also correct. To put this in another way, if we model a component as L' instead of L, we make *fewer assumptions about its behaviour*.

Consider now client 1 in Fig 3. Its interface consists of req_1, gra_1, and lve_1. It is a common assumption in system design that each component of a system may interact via other components only via the interface that was specified in the architecture of the system. Therefore, the alphabet of client 1 is $\{\mathsf{req}_1, \mathsf{gra}_1, \mathsf{lve}_1\}$.

We require that every trace of client 1 must be a prefix of $(\mathsf{req}_1\mathsf{gra}_1\mathsf{lve}_1)^\omega$. That is, the client must not try to execute its visible actions in a wrong order. In general in process algebras, the responsibility of this issue may be left on the client or the server, or distributed between them. We could put the responsibility on the server, by extending the client with the construction in automata theory textbooks that extends the transition relation of a deterministic finite automaton from a partial function to a full function. Then the server would have to be designed so that it blocks the added transitions. This difference is not important for the purpose of our present study, and to avoid spending space on it, we put the responsibility on the client.

Our next observation is that client 1 must have no divergence traces. Letting a client diverge would mean letting it steal all processor time and thus prevent the server and opposite client from making progress. In Section 2 we argued that to solve the mutual exclusion problem, it is necessary to assume that a client will not stay in the critical section forever. We are now in a similar situation and make a similar conclusion.

As a consequence, every trace σ of the client must eventually lead to a state such that no output transition of the state is labelled with τ. That is, if σ is a trace of client 1, then (σ, A) is a stable failure at least when $A = \emptyset$. It follows from the definition of stable failures that if $(\sigma, A) \in Sf(L)$ and $B \subseteq A$, then $(\sigma, B) \in Sf(L)$. Therefore, for each trace σ of client 1, we have the problem of determining the maximal A's such that (σ, A) is a stable failure of client 1.

We again appeal to the fact that a client must not stay in the critical section forever. We have chosen that after gra_1, the next visible action of the client may only be lve_1. Together these mean that for the traces that end with gra_1, there is a unique maximal A which is $\{\mathsf{req}_1, \mathsf{gra}_1\}$.

Assume that client 1 has executed req_1 and the server has committed to give it access. The server is thus ready to execute gra_1 and not gra_2. If client 1 now refuses to execute gra_1, a correct mutual exclusion system cannot do anything else than deadlock sooner or later. It cannot diverge or choose to execute gra_2, because by the properties of the parallel composition operator, if these options were available now, they would be available also if client 1 were willing to execute gra_1, compromising the eventual access property towards client 1. We see that it is not only the critical section where the client must be assumed to not stop. Indeed, we pointed out in Section 2 that assuming weak fairness on a transition is the norm and not assuming is exceptional. We conclude that for the traces that end with req_1, there is a unique maximal A which is $\{\mathsf{req}_1, \mathsf{lve}_1\}$.

Similar reasoning does not apply to the empty trace and the traces that end with lve_1. We concluded in Section 2 that the client must be given the permission to not execute req_1. So in this case, the maximal A is $\{\mathsf{req}_1, \mathsf{gra}_1, \mathsf{lve}_1\}$.

It follows from the definitions that every finite prefix of an infinite trace is a trace. In the case of client 1, the only infinite sequence that has this property is $(\mathsf{req}_1\mathsf{gra}_1\mathsf{lve}_1)^\omega$. To avoid over-specification, the guiding principle is that if we do not know whether some behaviour must be banned, we must not ban it. If banning it is necessary, then verification will fail, and when analysing the reason for failure, we will find out that banning would have been necessary. Guided by this principle, we do not ban $(\mathsf{req}_1\mathsf{gra}_1\mathsf{lve}_1)^\omega$. That Fig. 2 passes verification demonstrates that it need not be banned.

Figure 3(Right) shows a client with precisely the traces, etc., discussed above. It is optimal for the verification of eventual access: if the client may have more behaviour, then mutual exclusion cannot be solved, and if the client has less behaviour, then it has been over-specified, running the risk of intuitively incorrect solutions pass formal verification. From the point of view of this section, the τ-transition from state 1 to state 2 is unnecessary; the req_1-transition could start at state 1. The motivation of the τ-transition will be discussed in Section 4.

Figures 1 and 2 do not conform to the architecture in Fig. 3(Left). However, this is not a problem. Consider the system

$$\text{client 1} \parallel \text{server 1} \parallel \texttt{flag[1]} \parallel \texttt{flag[2]} \parallel \texttt{turn} \parallel \text{server 2} \parallel \text{client 2}$$

where the clients are like in Fig. 3(Right); the servers are like in Fig. 1 with communication with the clients added; and `flag[1]`, `flag[2]`, and `turn` model the variables in Fig. 1 in a standard fashion used in process algebras to model shared variables. Because of compositionality, this system can be recast as

$$\text{client 1} \parallel (\text{server 1} \parallel \texttt{flag[1]} \parallel \texttt{flag[2]} \parallel \texttt{turn} \parallel \text{server 2}) \parallel \text{client 2}$$

making it match Fig. 3(Left). It can also be recast as

$$(\text{client 1} \parallel \text{server 1}) \parallel \texttt{flag[1]} \parallel \texttt{flag[2]} \parallel \texttt{turn} \parallel (\text{server 2} \parallel \text{client 2})$$

```
1   bool req[2], gra[2], turn;
2   byte ncrit;               // nr of procs in critical section
3   active [2] proctype client()
4   {
5     do
6     ::  break;
7     :: // skip
8        // (!gra[_pid]);
9         req[_pid] = 1;
10        (gra[_pid]);
11        ncrit++; assert(ncrit == 1); ncrit--;  // critical section
12        req[_pid] = 0;
13    od
14  }
15  active proctype server()
16  { end: do                    // ok for the server to be blocked here
17    ::  (req[0] == 1 && (req[1] == 0 || turn == 0)) -> gra[0] = 1;
18        (req[0] == 0) -> gra[0] = 0; turn = 1;
19    ::  (req[1] == 1 && (req[0] == 0 || turn == 1)) -> gra[1] = 1;
20        (req[1] == 0) -> gra[1] = 1; turn = 0;
21    od
22  }
```

Fig. 4. A client–server mutual exclusion system with handshake

which can be transformed to Fig. 2 by computing (client 1 ‖ server 1) and (server 2 ‖ client 2), and then translating the system back to Promela.

4 Unprevented Request

In addition to under-specification of the requirements and over-specification of the clients, there is a third way in which an intuitively badly incorrect system may pass verification.

The model in Fig. 4 fails mutual exclusion and SPIN finds the error. Assume that both clients execute line 9, the server executes line 17, and client 1 continues to the end of line 12. Then the server passes the guard `req[0] == 0`, and is thus now at `->` on line 18. If client 1 acts fast, it can execute lines 9, 10 and 11 a second time before the server continues. Then the server completes line 18 and executes line 19, letting also client 2 continue to line 11. Now both clients are in the critical section.

The problem is that client 1 re-entered the critical section on the basis of the permission that it was given in the previous time, before the server had switched that permission off. This is a well-known problem and is solved by making the client wait until the previous permission has been switched off [3, p. 332]. This can be implemented by removing the comment symbol on line 8. After this modification, SPIN reports no error.

However, if also the comment symbol on line 7 is removed, then SPIN reports an invalid end state. Line 20 contains a bug. It assigns 1 to `gra[1]`, while it should assign 0 to it. As a consequence, after visiting the critical section once, client 1 can never again pass line 8. This is clearly unacceptable, so it is good that SPIN detects it. (Needless to say, the termination branch on line 6 is necessary for detecting the bug. This further illustrates the benefit of termination branches.)

The problem is that the error was *not detected* while the `skip` statement on line 7 was commented out. This illustrates another issue that we call *unprevented request*. In the absence of the `skip` statement, the system does not fail to serve the second, third, and later requests by client 1, for the vacuous reason that client 1 does not make such requests. It cannot, because it cannot pass line 8. So the system is formally correct, although it is intuitively unacceptable. Unprevented request means that a client must be able to freely choose whether to issue a request, without being prevented by the rest of the system.

As a matter of fact, if eventual access is expressed as $\Box(P_1 = 10 \rightarrow \Diamond(P_1 = 11))$, then it holds vacuously also in the presence of the `skip` statement. Then the model fails unprevented request because of line 8. Therefore, to enforce unprevented request, we must express eventual access as $\Box(P_1 = 8 \rightarrow \Diamond(P_1 = 11))$. This works in the presence of the `skip` statement. In its absence it does not work, because then SPIN treats lines 7 and 8 as the same state, making it possible for the client to execute line 6, contradicting the idea that it had requested.

The `skip` statement is thus necessary to avoid unprevented request. The analogue of the `skip` statement in Fig. 3(Right) is the τ-transition from state 1 to state 2. It is this transition that expresses in the model that the client has decided to seek access, and the next transition simply communicates this request to the rest of the system. The first cannot be blocked by the system even if the latter can be.

After fixing the bug on line 20, the model with line 8 commented out fails and with line 8 present passes verification with SPIN, independently of the presence or absence of the `skip` statement.

Unfortunately, defining eventual access as $\Box(P_1 = 8 \rightarrow \Diamond(P_1 = 11))$ introduces a problem. We will discuss it in the next section.

5 AG EF Intended Termination

Consider Fig. 4 after fixing line 20, without the comment symbols on lines 7 and 8, and with eventual access defined as $\Box(P_1 = 8 \rightarrow \Diamond(P_1 = 11))$. In the absence of fairness assumptions, the system has the execution where client 0 stays on line 8 or 9 while client 1 repeatedly visits the critical section. The server repeatedly serves client 1, because it is not aware of the request by client 0 before the latter has executed line 9. We see that to verify the formula, it is necessary to assume weak fairness, because otherwise the formula does not hold.

On the other hand, this model did pass verification in the previous section, although weak fairness was not used. This is because eventual access was not

checked. Instead, it was checked that the model does not deadlock when the clients have not executed their termination branches.

This illustrates that catching errors as unexpected deadlocks is not sensitive to the nuances of the formalization of eventual access as an $\mathrm{LTL}_{\mathsf{X}}$ formula. To the extent that it works, it works without any formalization. It suffices to specify the states where each process is allowed to be when the system terminates. This is often easy, because the default conventions of SPIN and Promela do much of the job. (In Fig. 4 we added `end:` on line 16 for this purpose.) This method specifies progress in general (to the extent it specifies it), instead of specifying one or more particular progress properties. This is an advantage for inexperienced users of $\mathrm{LTL}_{\mathsf{X}}$. On the other hand, as modification D in Section 2 demonstrates, not all important progress errors can be caught as unexpected deadlocks.

In [7, 13], a theory of *fair testing* was developed that facilitates an intermediate approach between detecting errors as unintended deadlocks and with standard $\mathrm{LTL}_{\mathsf{X}}$-based methods. If it is possible to reach a state from which a desired action d is not reachable, then both $\mathrm{LTL}_{\mathsf{X}}$ and fair testing declare that progress was violated. If all paths eventually lead to d, then both declare that progress holds. In the remaining case, there is an infinite path where d does not occur, but repeatedly an alternative path is available that leads to the occurrence of d. Fair testing declares this as progress and $\mathrm{LTL}_{\mathsf{X}}$ as non-progress. In terms of the Computation Tree Logic, **AG AF** d expresses progress in the $\mathrm{LTL}_{\mathsf{X}}$ sense while **AG EF** d expresses progress in the fair testing sense.

Fair testing does not need explicit formulation of fairness assumptions. It gives a weaker notion of progress than $\mathrm{LTL}_{\mathsf{X}}$, but it is much better than nothing. Checking it from the state space is technically simpler than that of $\mathrm{LTL}_{\mathsf{X}}$. It is exceptionally well suitable to be used together with stubborn set / partial order methods for alleviating the state explosion problem [12].

A generic progress requirement can be stated as *in all futures always, there is a future where eventually all processes are in a legal termination state.* This idea reduces the catching of progress errors to catching terminal strong components of the state space where some process is never in a legal termination state. If all such components happen to be deadlocks, we are back in catching errors as unexpected deadlocks.

6 Conclusions

We argued that a verification model should exhibit unforced request and unprevented request, and this can be obtained by adding termination branches and commitment to request similarly to Fig. 3(Right) and lines 6 and 7 in Fig. 4. Doing so widens the set of progress errors that are caught by catching unexpected deadlocks. This is an advantage, because this method does not need formulating fairness assumptions, and, being technically simple, deadlock detection is available in many tools. Furthermore, the method is compatible with the stubborn set method of alleviating state explosion, as explained in [12].

On the other hand, the method cannot catch all progress errors that can be caught with the standard method based on $\mathrm{LTL}_{\mathsf{X}}$ and fairness assumptions. Our method is useful when the standard method is considered too complicated. Furthermore, our observations on the importance of unforced request and unprevented request are worth considering also in the $\mathrm{LTL}_{\mathsf{X}}$ context.

We observed that such modelling style is rare. We do not interpret this as a sign of it not being worth using, but as a sign of its benefits not being known, although the idea has been in the literature for decades [10].

References

1. Victor Dyseryn, Rob J. van Glabbeek, and Peter Höfner. Analysing mutual exclusion using process algebra with signals. In Kirstin Peters and Simone Tini, editors, *Proceedings of EXPRESS/SOS 2017, Berlin, Germany, 4th September 2017.*, volume 255 of *EPTCS*, pages 18–34, 2017.
2. Roope Kaivola and Antti Valmari. The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In Rance Cleaveland, editor, *Proceedings of CONCUR '92, Third International Conference on Concurrency Theory*, volume 630 of *LNCS*, pages 207–221. Springer, 1992.
3. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.
4. Radek Pelánek. Beem: Benchmarks for explicit model checkers. In *International SPIN Workshop on Model Checking of Software*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.
5. Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
6. Glenn E. Reeves. What really happened on Mars?, 1997. https://www.cs.unc.edu/%7eanderson/teach/comp790/papers/mars_pathfinder_long_version.html [Online; accessed 7-May-2018].
7. Arend Rensink and Walter Vogler. Fair testing. *Inf. Comput.*, 205(2):125–198, 2007.
8. Antti Valmari. The weakest deadlock-preserving congruence. *Inf. Process. Lett.*, 53(6):341–346, 1995.
9. Antti Valmari. A chaos-free failures divergences semantics with applications to verification. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford–Microsoft Symposium in honour of Sir Tony Hoare*, Cornerstones of Computing, pages 365–382. Palgrave, 2000.
10. Antti Valmari and Manu Setälä. Visual verification of safety and liveness. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Proceedings*, volume 1051 of *LNCS*, pages 228–247. Springer, 1996.
11. Antti Valmari and Martti Tienari. Compositional failure-based semantics models for basic LOTOS. *Formal Asp. Comput.*, 7(4):440–468, 1995.
12. Antti Valmari and Walter Vogler. Fair testing and stubborn sets. *STTT*, 2017. https://doi.org/10.1007/s10009-017-0481-2.
13. Walter Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*, volume 625 of *LNCS*. Springer, 1992.