Stop It, and Be Stubborn!

Antti Valmari
Department of Mathematics
Tampere University of Technology
Tampere, Finland
Email: Antti.Valmari@tut.fi

***Abstract*—A system is always may-terminating, if and only if from every reachable state, a terminal state is reachable. This publication argues that it is beneficial for both catching non-progress errors and stubborn, ample, and persistent set state space reduction to try to make verification models always may-terminating. An incorrect mutual exclusion algorithm is used as an example. The error does not manifest itself, unless the first action of the customers is modelled differently from other actions. An appropriate method is to add an alternative first action that models the customer stopping for good. This method typically makes the model always may-terminating. If the model is always may-terminating, then the basic strong stubborn set method preserves safety and some progress properties without any additional condition for solving the ignoring problem. Furthermore, whether the model is always may-terminating can be checked efficiently from the reduced state space.**

***Index Terms*—model checking; stubborn set / partial order methods; safety; progress**

I. Introduction

Reduced state space construction using *ample* [2], *persistent* [6], or *stubborn* [14] sets works by, in each constructed state, computing a subset of transitions and only firing the enabled transitions in it instead of all enabled transitions. The subset is computed in a way that guarantees that correct answers to certain verification problems are obtained. The computation analyses the current state, static (that is, program-code-level) relations between the transitions, and perhaps also other available information. As a rule of thumb, the bigger the class of verification problems is, the fewer enabled transitions can be left out, and the bigger the resulting reduced state spaces are. Therefore, it often makes sense to design a method for a class of verification problems, even if it is a subclass of another class for which a method is already known.

Although the original publications on ample, persistent, and stubborn sets aimed at solving different verification problems and had significant differences in various details, it was obvious from the beginning that they are different members of the same family of methods. This family does not have a widely known unambiguous name. It is often called “partial order methods”, but equally often “partial order methods” refers to a larger family that also contains certain genuinely different methods, most importantly the unfolding method and sleep sets.

It has proven useful to investigate the methods on a more abstract level than on which the sets are actually computed. It facilitates combining ideas from all three original approaches and helps in developing new ideas. This more abstract view was developed gradually in several publications. It was presented in a fairly mature form in [14]. Since then, *static stubborn sets* have referred to definitions at the level on which the sets are actually computed, while *dynamic stubborn sets* have referred to the more abstract level.

When proving that a method preserves a certain class of properties, the corresponding dynamic definition is used as a starting point. When applying the method to a certain formalism for modelling systems, a static definition is provided and a theorem is proven saying that if the static definition holds, then also the dynamic definition holds. The algorithm for computing the sets is designed to yield sets that satisfy the static definition.

This publication works on this more abstract level. Therefore, the motivation and results of this publication apply to ample, persistent, and stubborn sets alike.

The most important example of a dynamic definition of stubborn sets consists of the conditions D0, D1, and D2 in Section IV. They constitute the *basic strong stubborn set method*. It guarantees that the full and reduced state spaces have precisely the same terminal states and that the reduced state space has an infinite execution if and only if also the full state space has.

The basic strong stubborn set method does not guarantee the preservation of safety properties such as mutual exclusion or liveness properties such as eventual access. This is because of the so-called *ignoring problem*. That is, the method may investigate a part of the state space that is unimportant for the safety or liveness property, find an infinite execution there, conclude that the rest of the state space cannot contain terminal states, and stop.

To preserve safety properties, various additional conditions have been suggested. One possibility is to recognize the terminal strong components of the reduced state space and ensure that in each of them, every enabled transition is fired. To preserve liveness properties, a common strategy is to ensure that every enabled transition is fired in every cycle of the reduced state space.

These additional conditions are problematic in two respects. First, there is the general phenomenon that the more conditions there are, the more enabled transitions the stubborn sets contain, and the bigger the reduced state space becomes. Second, as was pointed out in [4], a condition may choose the states where it fires all enabled transitions in an unfortunate way, leading to the construction of many more states than would be needed. The well-known liveness condition in [2] suffers from this problem.

In the present publication, a stunningly simple solution to the ignoring problem is suggested, proven correct, and experimented with. It suffices for safety and some progress properties. It is: if the modeller tries to make the verification models *always may-terminating*, then no additional conditions are needed at all. A model is always may-terminating if and only if from every reachable state, a terminal state is reachable. In the well-known logic called CTL [3] it can be expressed as "**AG EF** termination". "Tries to make" refers to the fact that the modeller need not prove that the model is always may-terminating. Instead, the model checker tool checks whether it is. In other words, not being always may-terminating is considered an error, and the model checker is guaranteed to reveal it (unless another error stops it first).

Trying to make models always may-terminating is a more natural goal than it might first seem. Using Peterson's mutual exclusion algorithm for n customers [9] as an example, Section II demonstrates that naive modelling may lead to the loss of non-progress errors. It is justified in Section III that this problem can be solved by making the customers of the algorithm capable of choosing to terminate. This makes the model as a whole always may-terminating. That is, even forgetting about stubborn sets, to check the eventual access property, the model must be made always may-terminating (or some more complicated method such as a suitable weak fairness assumption must be used).

A counterexample in [14] leaves little hope of finding an essentially better condition for the full class of linear-time liveness properties than some variant of the cycle condition. Another difficulty stems from the fact that the validity of linear-time liveness properties often depends on so-called fairness assumptions. They may be problematic for the modeller. Furthermore, although some publications have combined ample, persistent, or stubborn sets with commonly used fairness assumptions, none of the combinations seems to be fully satisfactory [1].

In essence, a typical linear-time liveness requirement corresponds to the CTL formula **AG**($\varphi \rightarrow$ **AF** ψ), where φ denotes that a request for service is pending and ψ denotes receiving the service. That is, for every reachable state, if a request for the service is pending in the state, then *every* future of the system must eventually lead to a state where the service is received. For this property, it is not important whether φ denotes the requesting of the service or the existence of a pending request (that is, a request has been made but not yet served).

Some authors have advocated the use of the strictly weaker notion **AG**($\varphi \rightarrow$ **EF** ψ). That is, for every reachable state, if a request for the service is pending in the state, then *at least one* future of the system must eventually lead to a state where the service is received. Here it is important that φ denotes that the request is pending. The property does not necessarily guarantee that the service will be received, but it does guarantee that receiving the service is not impossible and will not become impossible. With this notion, fairness assumptions become unnecessary. It may or may not be a sufficiently stringent correctness property from the practical point of view, but certainly it is much better than nothing. For instance, a process-algebraic variant of this theme was presented in [10].

With always may-terminating systems, this weaker notion reduces to the requirement that no terminal state has unsatisfied service requests. This condition can be modelled as a check that is run on each terminal state. The present author believes that modellers will not find it difficult to formulate such checks.

This approach facilitates early on-the-fly detection of safety and non-progress errors. The basic stubborn set method suffices during state space reduction. The order in which the state space is constructed is left unspecified, making it possible to use breadth-first for short counterexamples. As described in Section V, the check that the system indeed is always may-terminating can be implemented as a postprocessing step that is performed only if no errors are revealed during state space construction. Doing it as a postprocessing step is one factor that facilitates early on-the-fly detection of safety and non-progress errors. Furthermore, it makes the algorithm applicable to not always may-terminating systems as a one-sided test that can reveal safety and non-progress errors but cannot demonstrate their absence.

In addition to this approach to progress, a subset of linear-time liveness properties is covered in Section V.

Section IV presents the necessary background on stubborn sets. The new theorems are developed and their implementation options are discussed in Section V. Experimental results obtained with a new state space tool that implements the approach are reported in Section VII. Section VI describes the choice of the stubborn sets in the experiments.

II. A Motivating Example

Consider one or more concurrent processes called *customers*, each of which has a distinguished piece of code called *critical section*. The purpose of a *mutual exclusion algorithm* is to ensure that at any instant of time, no more than one customer is in the critical section. The algorithm must have the *eventual access property*, that is, if any customer tries to enter the critical section, it eventually succeeds. Typically it is assumed that an atomic operation can access at most one shared variable, and only once. For instance, if `i` is a shared variable, then `++i` involves at least two atomic operations, one reading the original value of `i` and another writing the new value.

Peterson's algorithm is a famous algorithm for solving the mutual exclusion problem on this level of atomicity. In his original publication [9], Peterson first described his algorithm for two customers and then generalized it to n customers for an arbitrary fixed positive integer n. We call these algorithms "Peterson-two" and "Peterson-n", respectively.

```
/* protocols for $P_i$ */
for $j$ := 1 to $n - 1$ do
begin
    $Q[i]$ := $j$;
    $TURN[j]$ := $i$;
    wait until ($\forall k \neq i, Q[k] < j$) or $TURN[j] \neq i$
end;
Critical Section;
$Q[i]$ := 0
```

Fig. 1. Peterson's algorithm for n customers [9].

Figure 1 shows Peterson-n copied verbatim from [9]. It implements $n - 1$ *gates*. To go through gate j, customer i writes j to the shared variable $Q[i]$. Then it gives priority to other customers by writing its own number i to the $TURN$ variable of the gate. It can go through the gate when no other customer is trying to go through the same or further gate, or when some other customer comes to the same gate, changing the $TURN$.

Figure 2 shows a model of Peterson-n written for ASSET. ASSET is A State Space Exploration Tool that is based on presenting the model as a collection of C++ functions that obey certain conventions [15]. The model is checked by copying it to the file `asset.model` and then compiling and executing `asset.cc`. This approach facilitates very fast execution of the transitions of the model and makes the modelling very flexible, because most features of C++ are available. On the other hand, the modelling language does not always support intuition well. This problem could be solved by implementing a preprocessor tool that inputs some nice modelling language and outputs the input language of ASSET. At the time of writing, no such tool has been implemented. At present, ASSET is unsuitable for production use, but it can be used for making scientific experiments.

Variables that describe the state of the model must be of the special type `state_var`. The value of such a variable is an unsigned integer in the range $0, \dots, 2^b - 1$, where $b = 8$ by default but can be specified for each state variable or array of state variables individually.

The modelling of the shared array Q of Peterson-n is obvious in Figure 2. The shared array $TURN$ has been abbreviated to `T`. Because the input language of ASSET has no notion of local variables of processes, j and k have been modelled as arrays. That is, `j[`i`]` models the j of customer i, and similarly with `k[`i`]`. The variable `S[`i`]` keeps track of the local state of customer i. It can be thought of as a program counter.

When ASSET has found an error, it prints a counterexample in the form of a sequence of states that leads from the initial state to an error state. Depending on the type of the error, the counterexample may also contain a cycle of states where the system fails to make progress towards some desired situation. For this purpose, the model must contain a `print_state` function. The function in Figure 2 presents the local states of the customers as characters, to make the print-out easier to interpret. Other than that, the function is straightforward.

The function `check_state` specifies the mutual exclusion property. The line `#define chk_state` commands ASSET to check every state that it has found by calling the `check_state` function. (It would have been nicer to use the same word `check_state` both in `#define` and as the function name. Unfortunately, C++ does not allow that.) By returning a character string, the function indicates that something is wrong with the state. This makes ASSET terminate the construction of the state space and print an error message that contains the string. That the state is good is indicated by returning the null pointer `0`.

The line `#define chk_may_progress` and the function immediately after it specify that for every state that the model can reach, the model may continue to a state where customer 0 is in the critical section. That is, the model cannot go into a state from which there is no path to a state where customer 0 is in the critical section. This represents the eventual access property. We call this particular form "may-access".

Peterson-two satisfies a stronger eventual access property which we call "must-access". In it, after a customer has set its Q variable, every path in the state space eventually leads to a state where the customer is in the critical section. However, because the "$\forall k \neq i, Q[k] < j$" test in Peterson-$n$ accesses more than one shared variable, and because one atomic operation may access at most one shared variable, the test must be implemented as a loop. An unsuccessful test introduces a cycle in the state space that does not take the customer to the critical section. That is, Peterson-n does not satisfy must-access. If must-access is specified and there are at least two customers, then ASSET reports an error. This is why may-access is used in Figure 2.

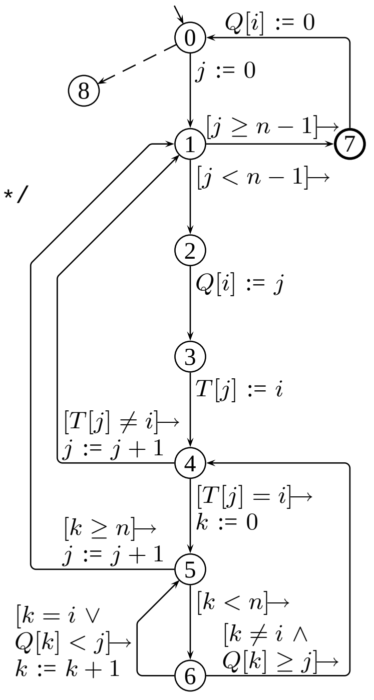
ASSET calls the function `nr_transitions` to find out how many transitions the model contains. The model in Figure 2 has one transition for each customer. It models all atomic operations of the customer. The grouping of atomic operations to transitions for ASSET is rather flexible. The only strict rule is that if two atomic operations may be executed in the same state and they yield different states, then they must belong to different transitions. This is because for ASSET, transitions must be deterministic. (This implies that a nondeterministic atomic operation must be modelled with more than one transition.)

Finally, the function `fire_transition` specifies the transitions. Given the number of a transition, it must either return `false` indicating that the transition is disabled in the current state, or modify the state according to the effect of the execution of the transition and return `true`. If it returns `false`, then it must not modify the state.

To improve readability, Figure 2 introduces a `goto(x)` macro. It moves the customer to local state `x` and indicates that the transition was enabled.

The modelling of the atomic operations $Q[i]$:= j, $TURN[j]$:= i, and $Q[i]$:= 0 of Peterson-n is straightforward. The **for** j loop has been modelled by cases 0 and 1 and the `++j[i];`

```
const unsigned n = 3;   // number of customers

state_var
  S[n],   // state of customer i: 0 = idle, 7 = critical, 1...6 = trying
  j[n],   // local variable j of customer i
  k[n],   // local variable k of customer i
  Q[n],   // number of gate through which customer i wants to go
  T[n-1]; // number of customer who has _no_ priority at gate j

const char ltr[] = { ’-’, ’j’, ’Q’, ’T’, ’w’, ’k’, ’A’, ’*’ };
void print_state(){
  for( unsigned i = 0; i < n; ++i ){
    std::cout << j[i] << ltr[ S[i] ] << k[i] << Q[i] << ’ ’;
  }
  for( unsigned i = 0; i < n-1; ++i ){ std::cout << T[i]; }
  std::cout << ’\n’;
}

/* Check that at most one customer is in critical section at any time */
#define chk_state
const char *check_state(){
  unsigned cnt = 0;
  for( unsigned i = 0; i < n; ++i ){ if( S[i] == 7 ){ ++cnt; } }
  if( cnt >= 2 ){ return "Mutex violated"; }
  return 0;
}

/* Check that customer 0 may always make progress. */
#define chk_may_progress
bool is_may_progress(){ return S[0] == 7; }

unsigned nr_transitions(){ return n; }

bool fire_transition( unsigned i ){
  #define goto(x){ S[i] = x; return true; }
  switch( S[i] ){
  case 0: j[i] = 0; goto(1)
  case 1: if( j[i] >= n-1 ){ goto(7) }else{ goto(2) }
  case 2: Q[i] = j[i]; goto(3)
  case 3: T[j[i]] = i; goto(4)
  case 4: if( T[j[i]] != i ){ ++j[i]; goto(1) }else{ k[i] = 0; goto(5) }
  case 5: if( k[i] >= n    ){ ++j[i]; goto(1) }else{ goto(6) }
  case 6: if( k[i] == i || Q[k[i]] < j[i] ){ ++k[i]; goto(5) }else{ goto(4) }
  case 7: /* critical section */  Q[i] = 0; goto(0)
  default: err_msg = "Illegal local state"; return false;
} }
```



$Q[i] := 0$
0
8
$j := 0$
$[j \geq n-1]\mapsto$
1
7
$[j < n-1]\mapsto$
2
$Q[i] := j$
3
$T[j] := i$
$[T[j] \neq i]\mapsto$ $j := j+1$
4
$[T[j] = i]\mapsto$ $k := 0$
$[k \geq n]\mapsto$ $j := j+1$
5
$[k < n]\mapsto$
$[k = i \vee Q[k] < j]\mapsto$ $k := k+1$
$[k \neq i \wedge Q[k] \geq j]\mapsto$
6

Fig. 2. A questionnable model of Peterson’s algorithm for n customers as ASSET code, and customer i as a state machine.

`goto(1)` in cases 4 and 5. In Figure 1, arrays are indexed starting from 1. However, consistently with the usual C++ convention, indexing starts from 0 in Figure 2. Cases 4 to 6 model the line

wait until $(\forall k \neq i, Q[k] < j)$ **or** $TURN[j] \neq i$

with two loops. The inner loop tries $k = 0$, $k = 1$, and so on until the $\forall$ test is found to fail or pass. The outer loop first tests whether $TURN[j] \neq i$ and if it fails, then starts the inner loop. If either test passes, then `++j[i]; goto(1)` is executed, modelling the completion of an iteration of the **for** j loop. If the $\forall$ test fails, then the outer loop is started again. Because the model checking is obviously incomplete if each customer tries only once to go to the critical section, a jump to the beginning has been added to case 7. The `default` branch is never entered, but the compiler complains if it is absent.

The entry “plain not non-progress revealing” in Table I in Section VII shows the state space size and state space construction and exploration time in seconds of this model. ASSET reported no errors. Excluding small variation in the times, the results remained the same when any customer replaced customer 0 in `is_may_progress`. However, the model was deemed “questionnable” in the caption of Figure 2. The reason for this will be discussed next.

In a second model of Peterson-n, customers were made able to not try to go to the critical section. To do this, a new local state 8 and a new transition to it from local state 0 were added to the model. When in local state 0, the customer chooses nondeterministically and without being affected by the other customers to either terminate (by going to local state 8) or to start trying to go to the critical section (by executing `j[i] = 0;` and going to local state 1). This was implemented by adding `' '` to the end of `ltr`; making `is_may_progress` to return `true` if and only if `S[0] >= 7`; making `nr_transitions` return `2*n`; adding `case 8: return false;` to the `switch` statement; and adding the following lines immediately before the `switch` statement:

```
if( i >= n ){
  i -= n;   // extract customer number
  if( S[i] == 0 ){ goto(8) }
  return false;
}
```

When $n = 2$, ASSET gives the following error message after 2.3 s of compilation and negligible analysis time:

```
0-00 0-00 0
0-00 0 00 0
==========
0j00 0 00 0
0Q00 0 00 0
0T00 0 00 0
0w00 0 00 0
----------
0k00 0 00 0
0A00 0 00 0
0k10 0 00 0
0A10 0 00 0
0w10 0 00 0
!!! May-type non-progress error
163 states, 326 arcs
```

In it, customer 1 terminates (its local state changes from "`-`" to " ") and then customer 0 goes to local state 1 ("`j`"). The line `==========` indicates that there is no path from this state to a state where customer 0 is in local state 7 or 8. This means that *customer 0 cannot go to the critical section after customer 1 has terminated*. So eventual access fails even in its less stringent form "may-access".

In the continuation of the counterexample, customer 0 goes to local state 4 as expected. Then it starts to run around a loop where it first executes cases 5 and 6 with $\mathtt{k} = 0$, then it executes them with $\mathtt{k} = 1$, and then goes back to local state 4. This corresponds to being stuck in the statement

$$\textbf{wait until } (\forall k \neq i, Q[k] < j) \textbf{ or } TURN[j] \neq i$$

Why does the customer not go through the gate? The part $TURN[j] \neq i$ does not let it pass, because it can be seen from the state that $\mathtt{T[0]} = 0$. Neither does the $\forall$ part, because $\mathtt{i} = 0$, $\mathtt{Q[1]} = 0$, and $\mathtt{j[0]} = 0$. In Peterson-n, $Q[1]$ would be 0 but j would be 1, because the **for**-loop starts with $j = 1$ in it. In Figure 2 indexing and thus also the **for**-loop were made to start with $j = 0$. This made the value 0 in entries of Q incorrectly mean both that the customer is not trying to go to the critical section and that it is trying to go through or has just gone through the first gate. It looks to customer 0 like the terminated customer 1 were trying to go through the first gate. Because customer 0 does not have priority, it keeps on running around the waiting loop.

Why does the error not manifest itself in the original model? In it, if customer i_1 is waiting at the first gate, some other customer i_2 eventually arrives at the same gate. This is because always at least one customer can make progress (the `T` test blocks at most one customer per gate and there are one fewer gates than customers), a progressing customer eventually reaches local state 0, and when all progressing customers are there, there is nothing else that the model can do than move one of them to the first gate. When arriving there, it gives priority to customer i_1 by assigning i_2 to `T[0]`.

To fix the error, `Q[i] = j[i]` in case 2 was changed to `Q[i] = j[i]+1` and the latter test in case 6 was changed to `Q[k[i]] <= j[i]`. The entry "correct" in Table I shows the results with the fixed model. No matter which customer was tested by `is_may_progress`, ASSET reported no errors.

To have an example of safety errors, finally the model was analysed that was obtained by swapping the statements `Q[i] = j[i]+1;` and `T[j[i]] = i;` in the correct model. ASSET reported a mutual exclusion error. In it, customer 1 went through the first gate while customer 0 stayed between the above-mentioned statements. Customer 1 got through, because `Q[0]` had not yet been assigned to. Then customer 0 got through because when customer 1 had passed by, it had given priority to customer 0 by assigning `1` to `T[0]`.

III. Appropriate Modelling of Not Requesting

The message of Section II is that

If, in a model, customers are not made able to not request for service, then non-progress errors may escape model checking.

Another way to look at this is that the first line in Figure 1 is different from the other places in that while in any of the latter, the customer must eventually go further if it can. Why the same must not be required of the first line was found out in Section II. It is obvious that if a customer never leaves the critical section, then the eventual access property cannot be provided to other customers without violating mutual exclusion. Although it is less obvious, a similar argument applies to most other places. For the remaining places the requirement is at least reasonable, even if not absolutely necessary.

In linear temporal logic [8], the customary way to express this difference is to assume so-called weak fairness towards all other transitions but not towards those that model the customers making requests. Every model has an imaginary "idling" transition that is always enabled. When the only thing that the modelled system can do is to request for the service, the model can avoid making the request by executing the idling transition.

Because the solution adopted in Section II is different, it is justified to ask whether it is appropriate. Certain process-

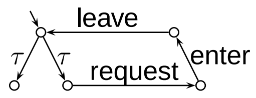


Fig. 3. A generic customer proper of a mutual exclusion system.

algebraic semantic theories provide strong evidence that it is appropriate. For the benefit of non-process-algebra-oriented readers, the discussion below is at the intuitive level.

The standard semantics of CSP [11], should testing [10], and the CFFD and NDFD semantics [17], among others, naturally yield a notion for deeming a process "better than" or "as good as" another process. The notion also applies to systems built as compositions of processes. If a component of a system is replaced by a "better" component, then the system as a whole either becomes "better" or remains "equally good". Within the limits of the information that is preserved by the semantics, if a system satisfies a specification, then also all "better" systems satisfy it. For instance, if a system satisfies a next-state-free linear temporal logic specification, then also each "NDFD-better" and each "CFFD-better" system satisfies it.

To apply this idea to Peterson-n, assume that each customer is split to two processes, a customer proper and a server. The customer proper is shown in Figure 3. The edges that are labelled with τ denote activities that are neither affected by nor observable by the rest of the model. The server is like in Figure 2. Request synchronizes with moving from local state 0 to local state 1 (that is, starting to try to go to the critical section), enter with the arrival to local state 7 (that is, arriving to the critical section), and leave with leaving local state 7.

This customer proper obviously does not do anything that it should not and does not stop when it should not. Furthermore, any tentative customer proper that is not "better than" or "as good as" Figure 3, tries to execute request, enter or leave when it should not; fails to execute request, enter or leave when it should; or, in the case of CSP, NDFD and CFFD, may steal all processor time at some point. So it is unacceptable. In other words, Figure 3 presents the "worst" acceptable customer proper, the one that makes the greatest challenge to the ability of the servers to guarantee mutual exclusion and eventual access. The system is correct with an arbitrary acceptable customer proper if and only if the system is correct with Figure 3. So the customer proper in Figure 3 is most appropriate for a verification model.

The parallel composition of customer proper with the server yields a process that plays the role of the original customer. With Figures 3 and 2, the result is like Figure 2 with its initial local state replaced by three local states and two `goto` commands. One of the three is the new initial local state, another is a terminal local state, and the third can be thought of as the initial local state of Figure 2. The `goto` commands lead from the new initial local state to the other two local states. They do not access any other variables than `S[i]`. The atomic operation that models leaving the critical section leads to the new initial local state instead of the initial local state of Figure 2.

Because the first atomic operation of Figure 2 does not access any shared variables, the above-mentioned semantic models see no difference if it is fused with the `goto` command from the new initial local state to the initial local state of Figure 2. Doing so yields precisely the second model of Section II.

Further discussion on this issue can be found in [16].

IV. Background on Stubborn Sets

We will need formal notation for concepts that have been used informally above.

The set of states is denoted with S and the set of transitions with T. In the case of ASSET, S consists of all possible combinations of values of the state variables, and $T = \{0, 1, \ldots, |T| - 1\}$, where $|T|$ is the number returned by `nr_transitions`.

That $t \in T$ is enabled at $s \in S$ is denoted by $s -t\rightarrow$, and if $\neg(s -t\rightarrow)$ holds, then t is disabled at s. The notation $s -t\rightarrow s'$ denotes that t is enabled at s and if t occurs (that is, is fired) at s, then the resulting state is s'. We assume that transitions are *deterministic*. That is, for any $s \in S$, $s_1 \in S$, $s_2 \in S$, and $t \in T$, if $s -t\rightarrow s_1$ and $s -t\rightarrow s_2$, then $s_1 = s_2$. A state is *terminal* if and only if no transition is enabled at it.

The obvious extension of $s -t\rightarrow s'$ to a finite sequence of transitions is denoted with $s -t_1 \cdots t_n\rightarrow s'$. State s' is *reachable* from state s if and only if there is a (possibly empty) sequence of transitions $t_1 \cdots t_n$ such that $s -t_1 \cdots t_n\rightarrow s'$. The initial state of the model is denoted with $\hat{s}$. A state is *reachable* if and only if it is reachable from $\hat{s}$.

The reachable states and the triples $s -t\rightarrow s'$ connecting them constitute a directed graph called *state space*. For this reason, $s -t\rightarrow s'$ is called *edge*. Also other directed graph terminology will be used, such as "path". The state space can be constructed by declaring $\hat{s}$ as found, and then firing, in each found state, all the transitions that are enabled in it. Each resulting state is declared as found. The algorithm is continued until all found states have been processed.

The *basic strong stubborn set method* assigns to each $s \in S$ a subset of transitions $\mathcal{T}(s) \subseteq T$, called *stubborn set*, such that the following conditions hold. In them, it is assumed that $t \in \mathcal{T}(s_0)$ and $t_1 \notin \mathcal{T}(s_0)$, ..., $t_n \notin \mathcal{T}(s_0)$.

D0: If s_0 is not terminal, then $\mathcal{T}(s_0)$ contains an enabled transition.

D1: If $s_0 -t_1 \cdots t_n t\rightarrow s'_n$, then $s_0 -t t_1 \cdots t_n\rightarrow s'_n$.

D2: If $s_0 -t\rightarrow$ and $s_0 -t_1 \cdots t_n\rightarrow s_n$, then $s_n -t\rightarrow$.

Many practical algorithms for constructing sets with the above properties have been presented. ASSET uses the strong component algorithm in [13]. To compute strong components, it uses the optimized version of Tarjan's algorithm [12] that has been presented in [5].

The *reduced state space* is constructed otherwise like the state space, but only the enabled transitions in $\mathcal{T}(s)$ are fired at s. The phrase *full state space* is a synonym of state space. It is useful when it may be unclear whether "state space" refers to the reduced or full state space. The sets of states and edges in the reduced state space are obviously subsets of the sets of states and edges in the full state space. We say that a state is an *r-state*, an edge is an *r-edge*, a path is an *r-path*, and so on, if and only if it is in the reduced state space. Obviously every r-state is a reachable state and every r-path is a path, but not necessarily vice versa.

The benefit of stubborn sets is that the reduced state space is often much smaller than the full state space, so its construction requires less time and memory. Even so, it can be used to check many properties of the model.

Letting $s = \hat{s}$ in the next theorem yields that the reduction preserves all reachable terminal states of the model, and for each path to a terminal state, the reduction preserves some permutation of it. Furthermore, an r-state is terminal if and only if it looks like terminal in the reduced state space. Although the theorem is old, its proof is presented here, because it is essential background to the new results in the next section.

Theorem 1 (old): Let s be an r-state and s_t be a terminal state such that $s \xrightarrow{t_1 \cdots t_n} s_\mathrm{t}$. The following hold.

- There is an r-path from s to s_t such that its sequence of transitions is a permutation of $t_1 \cdots t_n$.
- No r-edge starts at s_t.
- If s is an r-state and no r-edge starts at s, then s is terminal.

Proof: If $n = 0$, then the first claim is obvious. Otherwise $s \xrightarrow{t_1}$. By D0, some $t \in \mathcal{T}(s)$ is enabled at s. If none of t_1, …, t_n is in $\mathcal{T}(s)$, then D2 yields $s_\mathrm{t} \xrightarrow{t}$, contradicting the assumption that s_t is terminal. So there is $1 \leq i \leq n$ such that $t_i \in \mathcal{T}(s)$ and none of t_1, …, t_{i-1} is in $\mathcal{T}(s)$. By D1, there is s' such that $s \xrightarrow{t_i} s' \xrightarrow{t_1 \cdots t_{i-1} t_{i+1} \cdots t_n} s_\mathrm{t}$. The state s' is an r-state, and there is a path from s' to s_t of length $n - 1$. Repetition of this argument n times yields the first claim.

The second claim follows trivially from the fact that $\mathcal{T}(s_\mathrm{t}) \subseteq T$ and s_t is terminal. The last claim follows trivially from D0. ■

Also a lemma will be used in the sequel.

Lemma 2: If $t \in \mathcal{T}(s_0)$, $t_1 \notin \mathcal{T}(s_0)$, …, $t_n \notin \mathcal{T}(s_0)$, $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_n$, $s_0 \xrightarrow{t} s'_0$, and $s'_0 \xrightarrow{t_1} s'_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s'_n$, then for $1 \leq i \leq n$ we have $s_i \xrightarrow{t} s'_i$.

Proof: Let $1 \leq i \leq n$. By D2, there is s''_i such that $s_i \xrightarrow{t} s''_i$. By D1, $s_0 \xrightarrow{t t_1 \cdots t_i} s''_i$. Because transitions are deterministic, $s''_i = s'_i$. ■

V. Stubborn Sets on Always May-terminating Models

This section is devoted to new results that concern stubborn sets and always may-terminating models.

Theorem 3: The basic strong stubborn set method preserves the property "always may-terminating".

Proof: Assume first that the property holds on the full state space. That is, from every reachable state, a terminal state is reachable. Consider any r-state s. From it there is a path to a terminal state s_t. By Theorem 1, there is also an r-path from s to s_t, and s_t is terminal also in the reduced state space. So the property holds on the reduced state space as well.

Assume now that the full state space does not contain terminal states. Then by D0, the reduced state space does not contain terminal states either. So the property holds on neither state space.

Finally, assume that neither of the preceding cases holds. That is, the full state space contains a state from which no terminal state is reachable, but $\hat{s}$ is not such a state. We have to prove the existence of an r-state $\tilde{s}$ from which no r-terminal state is r-reachable. By Theorem 1 it suffices to prove that no terminal state is reachable from $\tilde{s}$.

For some natural number n and for $0 \leq i \leq n$, we will show the existence of a transition t_i, r-state s_i, states s'_i, s''_i, s^t_i, and finite sequences of transitions σ_i and ρ_i such that s^t_i is terminal, $s'_i \xrightarrow{t_i} s''_i$, $s_i \xrightarrow{\sigma_i} s'_i \xrightarrow{\rho_i} s^\mathrm{t}_i$, and there is no path from s''_i to a terminal state. Furthermore, if $i < n$, either σ_{i+1} is shorter than σ_i, or they are of the same length but ρ_{i+1} is shorter than ρ_i.

We choose $s_0 = \hat{s}$. Because the first case above does not hold, a state s''' is reachable from which no terminal state is reachable. Because the second case above does not hold, a terminal state is reachable from s_0. Therefore, along the path from s_0 to s''' there is an edge $s'_0 \xrightarrow{t_0} s''_0$ such that no terminal state is reachable from s''_0 and a terminal state which we call s^t_0 is reachable from s'_0. Then σ_0 and ρ_0 may be chosen such that $s_0 \xrightarrow{\sigma_0} s'_0 \xrightarrow{\rho_0} s^\mathrm{t}_0$. The base case has been proven.

To prove the induction step, we first consider the case where $t_i \notin \mathcal{T}(s_i)$. Similarly to the proof of Theorem 1, an application of D0 and D1 to the path $s_i \xrightarrow{\sigma_i} s'_i \xrightarrow{\rho_i} s^\mathrm{t}_i$ yields s_{i+1}, s'_{i+1}, $t'_{i+1} \in \mathcal{T}(s_i)$, σ_{i+1}, and ρ_{i+1} such that $s_i \xrightarrow{t'_{i+1}} s_{i+1} \xrightarrow{\sigma_{i+1}} s'_{i+1} \xrightarrow{\rho_{i+1}} s^\mathrm{t}_i$. Either $\rho_{i+1} = \rho_i$ and σ_{i+1} is obtained from σ_i by removing t'_{i+1}, or the same holds with the roles of σ and ρ swapped. In the former case, $s'_{i+1} = s'_i$ and we let $s''_{i+1} = s''_i$. Otherwise by Lemma 2 $s'_i \xrightarrow{t'_{i+1}} s'_{i+1}$, by D2 there is s''_{i+1} such that $s''_i \xrightarrow{t'_{i+1}} s''_{i+1}$, and by D1 and because t'_{i+1} and t_i are deterministic $s'_{i+1} \xrightarrow{t_i} s''_{i+1}$. No terminal state is reachable from s''_{i+1}, because otherwise a terminal state would be reachable from s''_i. The induction step is completed by choosing $s^\mathrm{t}_{i+1} = s^\mathrm{t}_i$ and $t_{i+1} = t_i$.

The case $t_i \in \mathcal{T}(s_i)$ remains. Then D1 can be applied to the path $s_i \xrightarrow{\sigma_i} s'_i \xrightarrow{t_i} s''_i$. If it picks a transition from σ_i, then the case is similar to the case $\rho_{i+1} = \rho_i$ above. Otherwise there is an r-state s_{i+1} such that $s_i \xrightarrow{t_i} s_{i+1} \xrightarrow{\sigma_i} s''_i$. If no terminal state is reachable from s_{i+1}, then it qualifies as $\tilde{s}$. Otherwise the same reasoning as in the base case with s_{i+1} playing the role of $\hat{s}$ and s''_i playing the role of s''' yields t_{i+1}, s'_{i+1}, s''_{i+1}, s^t_{i+1}, σ_{i+1}, and ρ_{i+1} with the required properties. The length claim holds, because σ_{i+1} is a proper prefix of σ_i.

Each step of the construction in the proof either yields $\tilde{s}$, shortens σ_i, or shortens ρ_i while retaining the length of σ_i. Because σ_i and ρ_i cannot become shorter without limit as i grows, eventually $\tilde{s}$ is obtained. ■

If the reduced state space is constructed in depth-first order, then it is possible to check efficiently on-the-fly that it is always may-terminating. By Theorem 3, the result applies also to the full state space. The check is based on computing the strong components of the reduced state space on-the-fly using Tarjan's algorithm [5], [12], recognizing terminal states, and propagating backwards the information whether a terminal state is reachable.

ASSET works in breadth-first order. So it cannot use this algorithm. Instead, it performs the check as a post-processing step. It re-constructs the edges storing them in reversed direction, and then performs a linear-time graph search starting at each terminal state.

The idea behind the implementation of stubborn sets in ASSET is that the modeller should try to make the model always may-terminating, but it is the responsibility of ASSET and not of the modeller to detect if it is not. The next three theorems list three properties that the basic strong stubborn set method preserves, if the model indeed is always may-terminating. For all of them, a counterexample found by the method is valid even if the model is not always may-terminating, but if the method finds no counterexamples, the result can be trusted only if the model is always may-terminating. Therefore, ASSET first checks the first two of them (the third one has not yet been implemented). If it finds no errors, it checks that the model is always may-terminating, giving an error message if it is not.

The following theorem tells that a well-known simple tool for checking linear-time safety properties works in our context.

Theorem 4: Assume that the model is always may-terminating. For any transition t_s, the basic strong stubborn set method preserves the property "t_s may become enabled".

Proof: If t_s cannot become enabled in the full state space, then clearly it cannot become enabled in the reduced state space either. If t_s may become enabled in the full state space, then there is a path $\hat{s} \xrightarrow{t_1 \cdots t_n} s_\mathsf{t}$ from the initial state to a terminal state such that $t_\mathsf{s} = t_i$ for some $1 \leq i \leq n$. By Theorem 1, t_s occurs also in the reduced state space. ■

If the construction of the reduced state space is aborted when t_s is found enabled, then t_s is never fired. In that case, t_s need not contain statements that change the state; the enabling condition suffices. Even so, to use Theorem 4, t_s must be taken into account in the construction of the stubborn sets. In ASSET, the enabling condition of t_s is represented via the `check_state` function.

To detect complicated errors, additional state variables that store some information about the history of the execution may be added to the model. For instance, consider a protocol whose purpose is to deliver messages from a sending site to a receiving site over an unreliable channel. To verify the ability of the protocol to prevent distortion of messages, when a message is given to the protocol, a copy of it is stored in an extra state variable. When the protocol delivers the message at the receiving site, `check_state` checks whether it is identical to what was stored in the extra variable.

The next theorem assumes that each state is classified as a progress state or other state. Typically progress states are those where the user either has not requested for service or has received the service that it requested. With this convention, a non-progress error occurs if and only if the user has requested for service but does not get it.

ASSET distinguishes between two types of progress states: *may* and *must*. In must progress, *every* path must lead to a progress state. If the state is terminal, then it must be a progress state in itself. This is the notion of progress typically used in linear temporal logic. In may progress, it suffices that *at least one* path leads to a progress state. Must progress implies may progress, but not necessarily vice versa. May progress is a branching-time property and related to the notion of home properties in Petri nets.

For reasons briefly mentioned in Section I, the stubborn set implementation of ASSET does not support must progress. The support of may progress is based on the following theorem.

Theorem 5: Assume that the model is always may-terminating. Its full state space contains a state from which no progress state is reachable if and only if it contains a terminal state that is not a progress state if and only if the reduced state space obtained with the basic strong stubborn set method contains such a state.

Proof: Assume that s is reachable but no progress state is reachable from it. Because the model is always may-terminating, a terminal state is reachable from s. It is a terminal state that is not a progress state. If a terminal state is not a progress state, then obviously no progress state is reachable from it. The first claim has been proven. The second claim follows from Theorem 1. ■

By the theorem, no other support for may progress would be needed in the case of always may-terminating systems than the `check_deadlock` feature of ASSET. Furthermore, it can be used for all customers simultaneously. However, when ASSET is used for other kinds of systems without stubborn sets, the notion of may progress states is useful. It is convenient that they can also be used with stubborn sets when they work with them.

The last theorem in this section can be used to check some linear-time liveness properties, such as "if the channel of a protocol passes (that is, does not lose) infinitely many messages, then the protocol as a whole passes infinitely many messages". Actually, it locates the challenge that linear-time liveness causes to stubborn sets precisely as the problem of preserving cycles that do not make progress.

Theorem 6: Let $t_\omega \in T$ and $T_* \subseteq T$. The basic strong stubborn set method on always may-terminating models preserves the property "there is an execution where t_ω occurs infinitely many times but no member of T_* occurs infinitely many times".

Proof: If such an execution exists in the reduced state space, then it is present also in the full state space.

Now assume that such an execution exists in the full state space. It is of the form $\hat{s} \xrightarrow{\rho} s_0 \xrightarrow{t_\omega \sigma_1} s_1 \xrightarrow{t_\omega \sigma_2} \ldots$, where no element of T_* occurs in any of the σ_i (but t_ω may occur in ρ). Let n_f be the number of states in the full state space, and let $m = 2n_\mathsf{f}^2$. There are $0 \leq j < k \leq n_\mathsf{f}$ such that

$s_j = s_k$. There are s_{t}, ρ_1, and ρ_2 such that s_{t} is terminal, $\hat{s} \,-\rho_1\!\!\rightarrow s_j \,-\rho_2\!\!\rightarrow s_{\mathrm{t}}$, $|\rho_1| < n_{\mathsf{f}}$, and $|\rho_2| < n_{\mathsf{f}}$.

So $\hat{s} \,-\rho_1\!\!\rightarrow s_j \,-(t_\omega \sigma_{j+1} \cdots t_\omega \sigma_k)^m\!\!\rightarrow s_j \,-\rho_2\!\!\rightarrow s_{\mathrm{t}}$. The application of Theorem 1 to it yields an execution in the reduced state space that contains at most $2n_{\mathsf{f}} - 2$ occurrences of elements of T_* and at least $2n_{\mathsf{f}}^2$ occurrences of t_ω. It has at least one part that contains at least n_{f} occurrences of t_ω and no occurrences of elements of T_*. Because the reduced state space contains no more states than the full, this part contains a cycle. The prefix of the execution up to the cycle together with an infinite number of repetitions of the cycle constitutes the infinite execution whose existence had to be proven. ■

This theorem does not facilitate meaningful use of Büchi automata with stubborn sets, because a Büchi automaton observes every action by the system. Thus it forces every enabled transition to every stubborn set, so the state space will not become smaller. For this reason, a related type of automata has been defined that only observes actions that may affect the validity of the property [7]. Unfortunately, some properties require the detection of cycles consisting solely of transitions that the automaton does not synchronize with. This seems to require the linear-time liveness cycle condition and perhaps also the representation of fairness assumptions as not part of the formula.

VI. Stubborn Sets in the Experiments

The construction of stubborn sets relies on rules of the form "if this transition is in the stubborn set of the current state, then also these other transitions must be". A complete implementation of stubborn sets would contain a preprocessor tool that reasons these rules from the model. Unfortunately, ASSET is not complete in this respect. As a consequence, the rules must be provided by the modeller. This is unacceptable from the point of view of industrial use. On the other hand, in Section I, the analysis of the program-code-level relations between the transitions was mentioned. Reduction results are very sensitive to the level of carefulness of this analysis. Because the rules may be provided by hand, ASSET facilitates easy experimenting with this issue.

Figure 4 shows the rules used in other experiments of this publication than the first. (In the first experiment, an obvious adaptation of the rules was used.) To discuss them, let s_0 denote the current state and $i \rightsquigarrow j$ denote that if transition number i is in the stubborn set $\mathcal{T}(s_0)$, then ASSET makes sure that also $j \in \mathcal{T}(s_0)$. It was mentioned after Theorem 4 that also the enabling condition of `check_state` must be taken into account. It will be discussed as a separate case and can be ignored until then.

The discussion below emphasizes the reasons why the rules are valid. So it gives an over-pessimistic impression of how difficult it is for a human or preprocessor tool to find the rules. Excluding case 4, all the rules arise from simple principles. Even for case 4, it is not beyond imagination that a preprocessor could find its rule.

Consider first the case $n \leq i < 2n$. Transition i models customer $i - n$ moving from the initial to the terminal local

```
void next_stubborn( unsigned i ){
  if( i >= n ){ stb(i-n); return; }
  switch( S[i] ){
  case 0: stb(i+n); return;
  case 1: return;
  case 2: stb_all(); return;
  case 3: stb_all(); return;
  case 4:
    if( T[ j[i] ] != i ){ return; }
    stb_all(); return;
  case 5: return;
  case 6: stb_all(); return;
  case 7: stb_all(); return;
  case 8: return;
  default: stb_all(); return;
  }
}
```

Fig. 4. Stubborn set rules for three models of Peterson-n.

state. Assume that it is the t of D1 and D2. So it is in $\mathcal{T}(s_0)$. The call `stb(i-n)` makes $i \rightsquigarrow i-n$ hold, implying that also transition $i-n$ is in $\mathcal{T}(s_0)$. It models all the remaining atomic operations of customer $i - n$.

If transition i is disabled, then D2 holds trivially. Furthermore, the only way to enable it is that customer $i - n$ moves to local state 0. Therefore, if $s_0 \,-t_1 \cdots t_n\!\!\rightarrow s_n$ and none of $t_1, \ldots, t_n$ is in $\mathcal{T}(s_0)$, then none of $t_1, \ldots, t_n$ is transition $i - n$, so transition i is disabled at s_n. This implies D1.

Assume now that transition i is enabled. Clearly the only way to disable it is that either it or transition $i - n$ occurs. So D2 holds. Because transition i does not access any variable that the other customers access, also D1 holds.

The case $0 \leq i < n$ remains. Transition i is disabled only in case 8. In that case nothing can enable it, so D1 and D2 hold independently of what other transitions are in $\mathcal{T}(s_0)$. Thus no rule of the form $i \rightsquigarrow j$ is needed. From now on we assume that transition i is enabled.

In cases 0, 1, and 5 transition i does not access variables used by other customers. We already say that transition $i + n$ never accesses variables used by other customers. So $\mathcal{T}(s_0) = \{i, i+n\}$ suffices, and no rule of the form $i \rightsquigarrow j$ where j refers to another customer is needed to make D1 and D2 hold. The rule $i \rightsquigarrow i + n$ can be dropped in cases 1 and 5, because then transition $i+n$ is disabled, so it does not matter whether $\mathcal{T}(s_0) = \{i, i+n\}$ or $\mathcal{T}(s_0) = \{i\}$.

In cases 2, 3, and 6 D1 and D2 are forced to hold by introducing a rule of the form $i \rightsquigarrow j$ for every transition j. As a consequence, no transition can be any of the $t_1, \ldots, t_n$ of D1 and D2.

The crucial observation behind case 4 is that the only transition that can turn `T[j[i]] != i` from `true` to `false` is i. The other customers may write to `T[j[i]]`, but they write to it their own index instead of i. Therefore, if `T[j[i]] != i`, then D2 holds automatically. D1 is not a problem either, because other than this test, the atomic operation does not access variables that are used by other customers. If `T[j[i]] == i`, then D1 and D2 follow like in cases 2, 3, and 6.

	plain			stubborn sets		
n	states	edges	time	states	edges	time
	not non-progress revealing					
2	133	266	0.0	88	124	0.1
3	38 038	114 114	0.3	18 817	34 083	0.2
4	12 346 971	49 387 884	70.3	4 312 993	8 988 034	22.2
	non-progress revealing					
2	163	326	0.1	116	162	0.0
3	43 675	131 025	0.3	23 134	41 562	0.2
4	14 186 506	56 746 024	85.6	5 316 461	10 903 336	36.9
	correct					
2	574	1 148	0.0	378	522	0.0
3	96 854	290 562	0.4	44 868	78 750	0.3
4	26 209 918	104 839 672	184	9 318 636	18 581 236	62.7
	mutex-violating					
2	336	602	0.0	219	258	0.0
3	32 957	87 081	0.2	15 164	22 100	0.2
4	6 614 675	23 547 787	16.4	2 116 738	3 527 255	5.8

TABLE I
RESULTS WITH ASSET ON MODELS OF PETERSON-n.

Case 7 is affected by `check_state`. (Without it, no rules would be needed.) Its rule comes from the principle that if a transition may change the state "further away" from the checked condition, then rules must be added to a "sufficient" subset of transitions that may change the state "closer to" the condition. Full formal treatment of this principle is beyond the scope of this publication but, to give some idea, let us show the correctness of this particular rule.

Let t_c be an imaginary transition such that its occurrence does not change the state and it is enabled if and only if the `check_state` in Figure 2 returns `true`. Case 7 may be interpreted as implementing the rules $i \leadsto t_\mathsf{c} \leadsto j$ for every transition j, forcing D1 and D2 to hold. It remains to be shown that excluding case 7, $i \leadsto t_\mathsf{c}$ is not needed. That is, we must show that t_c may be added to $T \setminus \mathcal{T}(s_0)$ without invalidating D1 and D2.

To prove that D2 remains valid, assume that $s_0 -t\rightarrow$ and $s_0 -t_1 \cdots t_\mathsf{c} \cdots t_n\rightarrow s_n$. Because t_c does not change the state, we have $s_0 -t_1 \cdots t_n\rightarrow s_n$. The assumption that D2 was valid beforehand yields $s_n -t\rightarrow$.

To prove that D1 remains valid, for some $0 \leq i \leq n$ let $s_0 -t_1 \cdots t_i\rightarrow s_i -t_\mathsf{c}\rightarrow s_i -t_{i+1} \cdots t_n\rightarrow s_n -t\rightarrow s'_n$. By D1 there are s'_0, ..., s'_{n-1} such that $s_0 -t\rightarrow s'_0 -t_1\rightarrow s'_1 -t_2\rightarrow \ldots -t_n\rightarrow s'_n$. Lemma 2 yields $s_i -t\rightarrow s'_i$. Because case 7 has been excluded from the discussion, the occurrence of t does not change the value of any `S[`i'`]` from 7 to something else. So $s_i -t_\mathsf{c}\rightarrow$ implies $s'_i -t_\mathsf{c}\rightarrow$. Because t_c does not change the state, we have D1.

VII. EXPERIMENTS AND DISCUSSION

Table I shows, for the models discussed in this publication and for different values of n, the number of reachable states, the number of edges in the state space (that is, successful transition firings), and the time it took to construct and explore the state space. The time is in seconds. In addition to the time in the table, a couple of seconds were spent on each model by the C++ compiler. The experiment was made on a Linux 1.6 GHz dual-core laptop with 2 GB of memory.

Because a safety error was detected in the mutex-violating model, the postprocessing steps that check progress properties and always may-termination were not executed. This explains the exceptionally short times obtained with the model.

A comparison of the results on the first two models tells that the addition of terminal states and transitions to them did not make the state space grow much.

ASSET has also an implementation of the well-known symmetry reduction method. However, the models discussed in this publication are not symmetric. Experiments have also been made with models where the $\forall$ test is represented as a single atomic operation. The symmetry method can then be used. Both methods together reduced the number of states of a deadlocking version to quadratic in n. In [15] an analysis of a demand-driven token-ring protocol was reported. With 10 processes, the plain method yielded 81 933 120 states in 262 seconds, stubborn sets yielded 1 514 900 states in 4.6 seconds, symmetries yielded 8 193 312 states in 59.5 seconds, and both together yielded 151 490 states in 0.9 seconds.

REFERENCES

[1] L. Brim, I. Černá, P. Moravec and J. Šimša: *On Combining Partial Order Reduction with Fairness Assumptions*. In: L. Brim et al. (eds.): FMICS and PDMC 2006, Lecture Notes in Computer Science 4346, Springer (2007) 84–99

[2] E.M. Clarke, O. Grumberg and D. Peled: *Model Checking*. The MIT Press (1999)

[3] E.A. Emerson: *Temporal and Modal Logic*. Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Elsevier Science Publishers (1990) 995–1072

[4] S. Evangelista and C. Pajault: *Solving the Ignoring Problem for Partial Order Reduction*. Software Tools for Technology Transfer 12(2) (2010) 155–170

[5] J. Eve and R. Kurki-Suonio: *On Computing the Transitive Closure of a Relation*. Acta Informatica 8(4) (1977) 303–314

[6] P. Godefroid: *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science 1032, Springer (1996)

[7] H. Hansen, W. Penczek and A. Valmari: *Stuttering-Insensitive Automata for On-The-Fly Detection of Livelock Properties*. In: R. Cleaveland and H. Garavel (eds.) Formal Methods for Industrial Critical Systems, Electronic Notes in Theoretical Computer Science 66(2) (2002) 178–193

[8] Z. Manna and A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer-Verlag (1992) 427 p

[9] G.L. Peterson: *Myths About the Mutual Exclusion Problem*. Information Processing Letters 12(3) (1981) 115–116

[10] A. Rensink and W. Vogler: *Fair Testing*. Information and Computation 205(2) (2007) 125–198

[11] A.W. Roscoe: *Understanding Concurrent Systems*. Springer (2010) 530 p

[12] R.E. Tarjan: *Depth-First Search and Linear Graph Algorithms*. SIAM Journal on Computing 1(2) (1972) 146–160

[13] A. Valmari: *Error Detection by Reduced Reachability Graph Generation*. Proceedings of the 9th European Workshop on Application and Theory of Petri Nets (1988) 95–122

[14] A. Valmari: *The State Explosion Problem*. In: W. Reisig and G. Rozenberg (eds.) Lectures on Petri Nets I: Basic Models, Lecture Notes in Computer Science 1491, Springer (1998) 429–528

[15] A. Valmari: *A State Space Tool for Models Expressed In C++ (tool paper)*. arXiv 1504:02597

[16] A. Valmari and M. Setälä: *Visual Verification of Safety and Liveness*. In: M.-C. Gaudel and J. Woodcock (eds.) Formal Methods Europe '96: Industrial Benefit and Advances in Formal Methods, Lecture Notes in Computer Science 1051, Springer (1996) 228–247

[17] A. Valmari and M. Tienari: *Compositional Failure-Based Semantic Models for Basic LOTOS*. Formal Aspects of Computing 7(4) (1995) 440–468