A State Space Tool for Concurrent System Models Expressed In C++

Antti Valmari

Department of Mathematics
Tampere University of Technology
P.O. Box 553, FI–33101 Tampere, FINLAND
Antti.Valmari@tut.fi

Abstract. This publication introduces a state space exploration tool that is based on representing the model under verification as a piece of C++ code that obeys certain conventions. This approach facilitates experimenting with many kinds of modelling ideas. On the other hand, the use of stubborn sets and symmetries requires that either the modeller or a preprocessor tool analyses the model at a syntactic level and expresses stubborn set obligation rules and the symmetry mapping as suitable C++ functions. The tool supports the detection of illegal deadlocks, safety errors, and may progress errors. It also partially supports the detection of must progress errors.

Keywords: model checking; stubborn sets; symmetries; safety; progress

1 Introduction

This publication discusses A State Space Exploration Tool called ASSET. It started in autumn 2014 as a simple quickly written tool that could be used to illustrate deadlocks, livelocks, the effect of the level of atomicity, and other concurrency-related issues to advanced programming students. The students had programmed in C++. An important goal was that starting to use the tool would not require complicated installation or learning a new language. Indeed, to install ASSET, it suffices to download one C++ file called `asset.cc`. (However, the need to learn concurrency-oriented *concepts* such as local state, global state and transition cannot, of course, be avoided.)

At the same time, the need arose to test a new method [10] of alleviating the state explosion problem. ASSET proved very suitable for that purpose.

ASSET is written in C++. From the point of view of its users, the key difference to other state space tools is that ASSET does not have an input language of its own, such as the Promela language of the SPIN tool [4] or the textual CSP language of the FDR tool [7]. Instead, the user represents the model under verification as a piece of C++ code that uses the pre-defined type `state_var` provided by ASSET and implements certain functions such as `fire_transition`, `print_state`, and `check_state`. The model is checked by copying it to the file `asset.model` and then compiling and executing the file `asset.cc`. The latter `#include`s the former.

The advantage is that the user can use most C++ features and his/her earlier programming knowledge very flexibly when writing models of concurrent systems and verification questions. The examples in this publication give an idea of what can be achieved. The disadvantage is that although the modelling of individual transitions is simple and natural, the part of code that directs execution to the right transition is often an ugly collection of `if`- and `switch`-statements.

This approach leads to very fast execution of the transitions of the model, because the model is compiled into machine code instead of being simulated by ASSET. This idea is not new. For instance, SPIN uses it.

Another drawback is that ASSET cannot perform any syntactic analysis on the model, because it is not read by ASSET but by the C++ compiler. This is not a big problem for plain state space exploration. On the other hand, some advanced methods such as the symmetry method [2, 3, 5] require such analysis.

The symmetry method relies on a function that, in a certain sense, represents the symmetry that is exploited. The typical approach is that the input language supports modelling the system in a way that makes the symmetry obvious, so that the tool can easily pick it and construct the necessary function. This means that also the modeller knows the symmetry. Designing the function only requires basic understanding of the symmetry method, and representing it in C++ is just a programming problem. ASSET is used in this way in Section 6. Although this is of course less convenient than the typical approach, it is realistic, especially when ASSET is used for teaching or for scientific experiments.

The advanced method known as stubborn sets [8–10] requires so complicated syntactic analysis that most users cannot be expected to perform it. However, it is reasonable to make experiments where an expert performs the analysis and writes the result in a form that is suitable for ASSET. The results of such experiments would tell whether it would be worth the effort to implement a preprocessor tool that would perform the analysis, or to implement the method in other state space exploration tools than ASSET. The experiment in [10] was of this kind, and another is reported in Section 5.

ASSET and many models are available at http://www.cs.tut.fi/~ava/ASSET/.

Throughout this publication, a *demand-driven token ring* is used as an example. It is introduced in Section 2 and modelled for ASSET in Section 3. Section 4 discusses the features that ASSET offers for specifying correctness properties. Sections 5 and 6 focus on the use of the stubborn set and symmetry methods in ASSET. Results of the experiments with the example system are collected in Table 1 towards the end of the publication. They clearly show the good time and memory efficiency of ASSET and the benefits of stubborn sets and symmetries.

2 The Demand-Driven Token Ring

The demand-driven token ring system is a mutual exclusion system. Its overall structure is shown in Figure 1. The system consists of n clients C_0, …, C_{n-1}

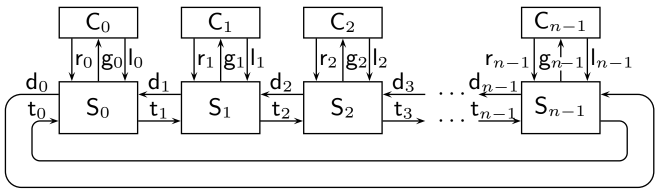

C_0
C_1
C_2
C_{n-1}
r_0 g_0 l_0
r_1 g_1 l_1
r_2 g_2 l_2
r_{n-1} g_{n-1} l_{n-1}
d_0 d_1 d_2 d_3 … d_{n-1}
t_0 t_1 t_2 t_3 … t_{n-1}
S_0
S_1
S_2
S_{n-1}

Fig. 1. Overall structure of the demand-driven token ring.

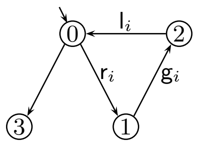
and n servers $S_0, \ldots, S_{n-1}$. Each client has a specific piece of code called *critical section*. The purpose of the system is to ensure that two or more clients are never simultaneously in their critical sections. This property is called *mutual exclusion*. The system must also guarantee *eventual access*, that is, always when a client has requested for access to its critical section, it eventually gets the permission to go there. The servers receive requests from the clients, grant permissions to enter the critical sections, and receive notices that the clients have left the critical sections. In Figure 1, these are denoted with arrows labelled with r_i, g_i, and l_i.

To guarantee mutual exclusion, precisely one *token* circulates in the ring. Only the server that has the token may grant permission to its client. To avoid unnecessary activity, the token is circulated only when there is a pending request. When a server S_i that does not have the token gets a request from its client, it demands the token from the previous server via d_i. The demand propagates in the ring until it reaches the server S_j that has the token. If C_j has made a request or is in its critical section, then S_j grants permission if it has not yet done that, and waits until C_j has left its critical section, after which S_j gives the token to $S_{j\oplus 1}$. By $x \oplus y$ we mean $(x + y)$ modulo n. Otherwise S_j just gives the token to $S_{j\oplus 1}$. The server $S_{j\oplus 1}$ first serves its own client if it has made a request, and then gives the token to $S_{j\oplus 2}$. In this way the token eventually propagates to S_i which can then grant permission to its server.

The clients are shown in state machine form in Figure 2 left. The critical section consists of state 2. In addition to the r_i-g_i-l_i cycle that models requesting for access and then entering and leaving the critical section, there is a transition from state 0 to state 3. It models the *unforced request* property, that is, that the client may choose to not request for access. This is important for catching certain kinds of errors.

To see this, assume that the servers are modified such that after getting the token, each server always waits until its client makes a request and then serves its client before giving the token to the next server. This is incorrect, because if the client never makes the request, then the token is blocked at that server, and the requests by other clients remain unsatisfied forever.

However, in the absence of the transition from state 0 to state 3, this error would not be caught. This is because most if not all formalisms for concurrency make implicitly or explicitly the assumption that if something can happen, then something will happen. (Without this assumption, systems could deadlock at any time without a reason, which of course would be an inappropriate model



```
0: wait until $\mathsf{r}_i$ or $\mathsf{d}_{i\oplus 1}$ has occurred
   if $\mathsf{t}_i$ has not occurred then $d_i$
   goto 1
1: wait until $\mathsf{t}_i$ has occurred
   if $\mathsf{r}_i$ has occurred then $\mathsf{g}_i$; goto 2
   else $\mathsf{t}_{i\oplus 1}$; goto 0
2: wait until $\mathsf{l}_i$ has occurred
   $\mathsf{t}_{i\oplus 1}$; goto 0
```

Fig. 2. The clients and servers.

of reality.) With the assumption, with the modified servers, and without the transition from state 0 to state 3, consider the situation where every client has made the request except the one whose server has the token, and all demands have propagated to that server. Then the only thing that can happen is that this client makes a request. The assumption forces it to happen. So the client is forced to make the request even if it does not want to. The client behaviour that would reveal that the servers are incorrect is thus left out from the analysis.

Another way to look at this is that there is a fundamental difference between the transitions from state 2 to state 0 and from state 0 to state 1. If the client chooses to stay in its critical section — that is, state 2 — forever, then requests by other clients cannot be satisfied without violating mutual exclusion. On the other hand, not satisfying them violates eventual access. Because of this, every reasonable analysis of eventual access assumes that if a client is in its critical section, it eventually leaves it. On the other hand, we just argued that we must not assume that if a client is in its initial state, it eventually makes a request. This is a fundamental difference.

Often this difference is represented with so-called idling transitions and fairness assumptions, as explained in [6]. Another mainstream method is to exploit some variant of so-called failures [1, 7]. We do so by adding the transition from state 0 to state 3. The client is not forced to take the transition from state 0 to state 1, because it can take the transition from state 0 to state 3 instead.

That this model is appropriate has been argued in [10,11], based on the theory of (stable) failures. We do not present full details here, but do briefly discuss one issue that has caused confusion. Because the client cannot come back from state 3 to state 0, this model might seem inappropriate in the situation where the client does not want to make a request now but wants to make it some later time. However, this situation is represented by the possibility of the client staying at state 0 until it wants to make a request again. Moving to state 3 only represents the possibility of making a request *never* again.

The behaviour of the servers is too tricky to be shown here as a state machine. Instead, it is represented in pseudocode in Figure 2 right. A precise ASSET model will be shown in Figure 4.

The server stays in state 0 as long as it has no request or demand to serve. While in this state, it may have the token. A request or demand makes it to move to state 1. When going there, it demands the token, if it does not yet have it. It

```
#ifdef size_par               // n is the number of clients and servers
const unsigned n = size_par;  // n comes from the compilation command
#else
const unsigned n = 6;         // default value of n
#endif

state_var
  C[n] = 2, // state of client i:
            // 0 = idle, 1 = requested, 2 = critical, 3 = terminated
  S[n] = 2, // state of server i:
            // 0 = idle, 1 = waiting for token, 2 = waiting for client
  T[n] = 1; // true <==> server i has token

#ifdef symm_must  // a trick used with the symmetry method
state_var c0now;  // the current index of the original client 0
#else
unsigned const c0now = 0;
#endif

const char Cchr[] = { '-', 'R', 'C', ' ' }, Schr[] = { 'i', 'w', 't' };
void print_state(){
  for( unsigned i = 0; i < n; ++i ){
    std::cout << Cchr[C[i]] << Schr[S[i]];
    if( T[i] ){ std::cout << '*'; }else{ std::cout << ' '; }
  }
  std::cout << '\n';
}
```

Fig. 3. Model of the demand-driven token ring, part 1.

stays in state 1 until it has the token. If it does not have a pending request by its client, it gives the token to the next server and returns to the initial state 0. Otherwise it grants the permission to its client and goes to state 2. It waits there until its client leaves the critical section, after which it gives the token to the next server and returns to the initial state.

When going from state 2 to state 0, the server gives the token to the next server even in the absence of a demand. The purpose of this is to ensure that if its own client makes immediately a new request, it is not served before the other clients have had the chance to go to the critical section.

3 ASSET Model of the System

Figures 3 and 4 show the model of the example system.

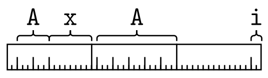
The `#ifdef size_par` structure makes it possible to specify the number of clients and servers via an option that is given to the C++ compiler. Also many features of ASSET can be controlled in a similar way. For instance, with the Gnu

C++ compiler, the options `-Dstubborn -Dsize_par=13 -Dstop_cnt=100000000` command ASSET to use stubborn sets, set $n = 13$, and stop the construction of the state space when 10^8 states are exceeded.

The initialization `C[n] = 2` does not specify that the initial local state of each client is 2 (the critical section) but that two bits are used for representing the local state of each client. The value of each state variable is an unsigned integer in the range $0, \ldots, 2^b - 1$, where b is the number of bits. The default value of b is 8. The initial value of each state variable is 0.

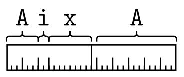
Internally, ASSET represents the state of the model as a sequence of unsigned integers. To save memory, ASSET packs many state variables into the same unsigned integer when possible. A `state_var` object does not store the value of the state variable, but information on in which unsigned integer and in which of its bits the value is stored. If the most recently employed unsigned integer has at least as many unused bits as the next state variable needs, then ASSET puts the state variable there. This implies that the order in which the state variables are declared may affect the amount of memory that ASSET uses per state. For instance, assuming 16-bit unsigned integers,

```
state_var x, A[7]=3, i(2);
```



consumes three unsigned integers, while

```
state_var x, i(2), A[7]=3;
```



consumes only two.

The purpose of `c0now` will be explained in Section 6. Until then, please assume that `c0now` is 0.

When ASSET has detected an error, it prints a counterexample in the form of a sequence of states. For this purpose, it needs a `print_state` function. To improve the readability of the counterexamples, the function in Figure 3 uses character encodings for local states. The use of C++ facilitates this and other methods for making the printout look natural for the model. This has been widely exploited in the models on the web page of ASSET.

The function `nr_transitions` in Figure 4 tells ASSET how many *transitions* the model contains. The most common case is that a transition models one or more atomic operations of the system. Transitions in ASSET are deterministic. This implies that nondeterministic operations such as tossing a coin must be modelled by more than one transition. Other than that, ASSET does not restrict the grouping of atomic operations to transitions.

In its initial state, a client of the example system makes a nondeterministic choice between requesting access and terminating for good. For this reason, two transitions are used to model each client. Each server has one transition.

The function `nr_transitions` may also be used for implementing whatever operations are necessary before starting the construction of the state space. In Figure 4, it is used for giving the token to client 1. Client 1 was chosen here, because eventual access will be tested for client 0, so we wanted it to lack the token initially.

```
unsigned nr_transitions(){ T[1] = true; return 3*n; }

inline unsigned next( unsigned i ){ return (i+1) % n; }
inline unsigned prev( unsigned i ){ return (i+n-1) % n; }

bool fire_transition( unsigned i ){

  /* Servers */
  if( i >= 2*n ){
    i -= 2*n;
    #define goto(x){ S[i] = x; return true; }
    switch( S[i] ){
    case 0:
      if( C[i] == 1 || ( S[next(i)] == 1 && !T[next(i)] ) ){ goto(1) }
      return false;
    case 1:
      if( !T[i] ){ return false; }
      if( C[i] == 1 ){ C[i] = 2; goto(2) }
      if( S[next(i)] == 1 ){ T[i] = false; T[next(i)] = true; goto(0) }
      return false;
    case 2:
      if( C[i] == 2 ){ return false; }
      T[i] = false; T[next(i)] = true; goto(0)
    default: err_msg = "Illegal local state"; return false;
    }
    #undef goto
  }

  /* Clients */
  #define goto(x){ C[i] = x; return true; }
  if( i >= n ){   // termination transition
    i -= n;
    if( C[i] == 0 ){ goto(3) }else{ return false; }
  }
  if( C[i] == 0 ){ goto(1) }  // request access
  if( C[i] == 2 ){ goto(0) }  // leave critical
  return false;
}
```

Fig. 4. Model of the demand-driven token ring, part 2.

The transitions of the model are numbered starting from 0. The function `fire_transition(`t`)` returns `true` or `false` to indicate whether transition number t is enabled. If t is enabled, then `fire_transition` modifies the state according to the occurrence of t. If t is disabled, then `fire_transition` must not modify the state. This rule makes it possible for ASSET to try the next transition without having to upload the state again.

To improve readability, Figure 4 introduces two versions of a `goto(x)` macro. They model the server and the client going to local state `x` and indicate that the transition was enabled.

If $0 \leq t < n$, transition t models client t going either from local state 0 to local state 1 (that is, requesting access) or from local state 2 to local state 0 (leaving the critical section). If $n \leq t < 2n$, transition t models client $t - n$ going from local state 0 to local state 3 (that is, terminating).

Finally, the transitions $2n \leq t < 3n$ model server $t - 2n$. It waits in local state 0 until its client requests access or the next server needs the token. For the reason discussed in Section 5, it tests that the next server does not already have the token.

Then it waits in local state 1 until it has the token. If its client has requested access, it moves the client to the critical section and goes to local state 2. Otherwise, if the next server needs the token, server $t - 2n$ gives it to it. Otherwise, server $t - 2n$ continues waiting.

In local state 2, server $t - 2n$ waits until its client has left the critical section. Then it gives the token to the next server and returns to the idle state. As a consequence, its client cannot get access again before the token has circulated through the ring and the other clients have had the chance to get access.

The `default` branch is explained in Section 4.

4 The Checking Features

Figure 5 shows the checking functions used in the experiments of this publication. Each of them can be switched off by commenting out the corresponding `#define`, without having to comment out the function as a whole. This is handy for experimenting. (It would have been nice to use the same word in the `#define` and as the name of the function, but C++ does not allow that.) More flexibility comes from the fact that if `xxx` has not been switched on in the model with `#define xxx`, then it can be switched on at compile time with a compiler option.

ASSET operates in stages. In the first stage, it checks for safety errors and illegal deadlocks (if the checking of them has been switched on). It constructs the state space in breadth-first order, to minimize the length of counterexamples. ASSET calls `check_state` each time when it has constructed a new state, and `check_deadlock` when it has tried to fire transitions in a state but none was enabled. If the state is not good, the function returns a character string. ASSET prints an error message containing it and terminates. That is, ASSET implements on-the-fly detection of safety and illegal deadlock errors. That the state is good is indicated by returning the null pointer `0`.

```
/* Check that at most one client 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( C[i] == 2 ){ ++cnt; } }
  if( cnt >= 2 ){ return "Mutual exclusion violated"; }
  return 0;
}

/* Check that every client has stopped. */
#define chk_deadlock
const char *check_deadlock(){
  for( unsigned i = 0; i < n; ++i ){
    if( C[i] != 3 ){ return "Client not terminated"; }
  }
  return 0;
}

/* Check that the original client 0 eventually gets access
  if it wants to. */
//#define chk_must_progress
bool is_must_progress(){ return C[c0now] != 1; }
```

Fig. 5. The check functions of the model of the demand-driven token ring.

By using a global or static C++ variable, `check_deadlock` (or `check_state`) can be made to count the number of deadlocks that it has detected, and only give the error message for, say, the tenth. Thus, although ASSET does not provide a mechanism for investigating each deadlock in turn, C++ makes it possible.

If ASSET did not detect any errors and if it has further checks to perform, it constructs a data structure that contains the edges of the state space in the reverse direction. To do that, it goes through all states that it has found and fires the transitions again in them. (In the case of stubborn sets, it fires the same subsets of transitions.) In this way, only one unsigned integer per edge is needed. Storing the edges during the first stage and sorting them afterwards would have used two unsigned integers per edge.

Then, if `chk_may_progress` is on, ASSET checks the state space for *may progress errors* by performing a linear-time search along the reversed edges. A may progress error is a reachable state from which no terminal state and no state accepted by `is_may_progress` is reachable. May progress can be thought of as a less stringent alternative to linear-time liveness that does not need fairness assumptions. This feature has been discussed extensively in [10] and is not used in the experiments of the present publication, so it will not be discussed further here.

Next ASSET checks the state space for *must progress errors*, if it has been commanded to do so and it has not yet terminated because of another error. A must progress error is a cycle in the state space that does not contain any state accepted by `is_must_progress`. This is a restricted form of checking linear-time liveness. For reasons discussed in the next two sections, the simultaneous use of this feature with stubborn sets and symmetries is limited. Therefore, it has been switched off in Table 1.

Outside Table 1, the `is_must_progress` function in Figure 5 was switched on in some experiments. ASSET found no errors in the model in Figures 3 and 4. Also a modified model was used where, when leaving local state 2, instead of giving the token to the next server and going to local state 0, the server goes to local state 1. ASSET reported that this model has a cycle where a requesting client does not get access. In it, another client leaves the critical section, requests for access again, and gets access again. The output of ASSET is shown below with explanatory comments. (ASSET does not minimize the length of counterexamples that contain a cycle.)

```
-i -i*       initial state
==========   after this line, must progress fails for a pending request
Ri -i*       client 0 requests
Rw -i*       server 0 waits for token
----------   after the last line, the counterexample jumps back here
Rw -w*       the demand reaches server 1
Rw Rw*       client 1 requests
Rw Ct*       client 1 gets permission
Rw -t*       client 1 leaves the critical section
!!! Must-type non-progress error
46 states, 66 edges
```

Finally, if the stubborn set method is used, safety or progress was checked, and ASSET has not yet found any error, it checks that the state space is always may-terminating in the sense discussed in the next section.

The model may at any time assign a character string to `err_msg`. It causes ASSET to terminate and print an error message containing the string. This feature is not intended for specifying correctness properties, but for catching inconsistent situations within the model. In Figure 4 it is used in the `default` branch of the `switch` statement, to indicate that the modeller believes that the branch is never entered.

In addition to the memory needed for the state itself, ASSET uses two or five unsigned integers per state, depending on whether the verification task involves graph search operations in the state space. To quickly detect whether a state is new or has been constructed also earlier, there is a hash table whose base table uses 2^h unsigned integers, where h is 23 by default but can be changed. The stubborn set method consumes an additional $5t$ unsigned integers, where t is the number returned by `nr_transitions`.

Therefore, in theory, ASSET uses approximately $((2+s)n+2^h)u$ bytes for the easier and $((5+s)n+m+2^h)u$ or $((5+s)n+m+2^h+5t)u$ bytes for the more difficult verification tasks, where n is the number of constructed states,

```
void next_stubborn( unsigned i ){

  if( i >= 2*n ){
    i -= 2*n;
    switch( S[i] ){
    case 0: if(
              C[i] == 1 || ( S[next(i)] == 1 && !T[next(i)] )
            ){ return; }
            stb(i, next(i)+2*n); return;
    case 1: if( !T[i] ){ stb(prev(i)+2*n); return; }
            if( C[i] == 1 ){ return; }
            if( S[next(i)] == 1 ){ stb(i); return; }
            stb(i, next(i)+2*n); return;
    case 2: if( C[i] == 2 ){ stb(i); }
            return;
    default: return;
    }
  }

  if( i >= n ){ stb(i-n); return; }
  switch( C[i] ){
  case 0: stb(i+n, i+2*n); return;
  case 1: stb(i+2*n); return;
  case 2: stb_all(); return;
  default: return;
  }

}
```

Fig. 6. The stubborn set obligation rules of the demand-driven token ring.

m is the number of constructed edges, s is the number of unsigned integers used for representing a state, and u is the number of bytes in an unsigned integer (usually $u = 4$ or $u = 8$). These formulae correctly predict the numeric and "–" entries in Table 1. However, because of how the dynamically growing arrays of C++ work, the real memory consumption may add almost $(2 + s)nu$ or $(5 + s)nu$ bytes to this. With 13 servers and clients, the stubborn set method yields $13 \cdot 3\,777\,949 = 49\,113\,337$ states. This would otherwise fit the memory, but the doubling of the arrays at $2^{25} = 33\,554\,432$ states causes a memory overflow, explaining the ".." entries in Table 1.

5 The Stubborn Set Method in ASSET

The implementation of the stubborn set method in ASSET is discussed extensively in [10]. Therefore, it is discussed here only briefly.

Only the basic strong stubborn set method that preserves the deadlocks is implemented. However, theorems in [10] tell that if the model is always may-terminating — that is, if from every reachable state, a terminal state is reachable, then the basic strong stubborn set method preserves also safety and certain progress properties. Furthermore, whether the model is always may-terminating can be checked from the reduced state space.

To use the method, the function `next_stubborn` must be provided. It represents state-dependent rules of the form "if this transition is in the stubborn set, then also these transitions must be". Figure 6 shows the rules used in the experiments reported in Table 1.

The present author did not at first realize the necessity of the part `&& !T[next(i)]` in case 0 in Figure 4. When it was lacking, the plain and symmetry methods did not give any error messages. Indeed, mutual exclusion and eventual access are not violated. However, thanks to the check that the model is always may-terminating, the stubborn set method gave the following when $n = 2$.

```
-i -i*
-i  i*
==========
Ri  i*
Rw  i*
Rw  w*
Rw* i
Rw* w
Ct* w
-t* w
-i  w*
----------
 i  w*
 w  w*
 w* i
 w* w
!!! State was reached from which termination is unreachable
67 states, 93 edges
```

In it, client 0 visits the critical section and both clients terminate. Because waiting information may be propagated from server $i \oplus 1$ to server i even if the former has the token, unnecessary waiting information enters the ring. Eventually the model runs in a cycle where unnecessary waiting information circulates in one direction and, driven by it, the token circulates in the opposite direction. The cycle consists of the states below the "`----------`" mark. Reaching a terminal state is impossible after the "`==========`" mark. So the first erroneous state is `Ri i*`.

The stubborn set implementation in ASSET does not guarantee that must progress errors are found. If must progress is used with stubborn sets and ASSET finds no errors, then it gives a warning that the pass verdict is unreliable. With the modified model discussed in Section 4, ASSET did find the error with stubborn sets switched on. With $n = 8$, there were 2 472 336 states and 17 539 200 edges without and 163 264 states and 293 984 edges with stubborn sets.

```
void symmetry_representative(){
  unsigned i = 0;
  while( !T[i] ){ ++i; }  // find the server with the token
  i = prev(i);
  if( !i ){ return; }     // terminate, if the state maps to itself
  unsigned A[n], j;
  for( j = 0; j < n; ++j ){ A[j] = C[(i+j) % n]; }
  for( j = 0; j < n; ++j ){ C[j] = A[j]; }
  for( j = 0; j < n; ++j ){ A[j] = S[(i+j) % n]; }
  for( j = 0; j < n; ++j ){ S[j] = A[j]; }
  for( j = 0; j < n; ++j ){ A[j] = T[(i+j) % n]; }
  for( j = 0; j < n; ++j ){ T[j] = A[j]; }
  #ifdef symm_must
  c0now = (c0now + n-i) % n;
  #endif
}
```

Fig. 7. The symmetry representative function of the demand-driven token ring.

6 The Symmetry Method in ASSET

Many systems contain similar components organized in a symmetric fashion. Several authors have suggested exploiting the symmetry for reducing the size of the state space, including [2, 3, 5].

The implementation of the symmetry method in ASSET is very simple. Unfortunately, as we will see, it leaves a lot of responsibility to the modeller or preprocessor tool.

Most importantly, the modeller or preprocessor must provide a function `symmetry_representative` that maps each state to a symmetric state. The more states are mapped to the same state, the better are the reduction results. Ideally, all states that are symmetric to each other are mapped to the same state. However, the method remains correct even if the function is not ideal in this respect.

If the symmetry method has been switched on, ASSET calls the function `symmetry_representative` on the initial state and on each result of a successful firing of a transition. As a consequence, paths in the reduced state space may contain *symmetry hops*, that is, the head state of an edge is not necessarily the real result of firing the transition in question in the tail state of the edge. Instead, it may be another state that is symmetric to the real result.

Figure 7 shows the symmetry mapping used in the experiments of this publication. It first finds the server that has the token, and then rotates the ring so that the found server becomes server 1.

The modeller or preprocessor must take the symmetry mapping into account when formulating the checked properties. Because each of `check_state` and `check_deadlock` analyses a single state, it is not difficult to make them give the same reply on symmetric states. The versions in Figure 5 do so.

It is more difficult with progress properties. For instance, consider the eventual access property "if client 0 wants to go to the critical section, it eventually gets there". When `symm_must` is off, the `is_must_progress` function in Figure 5 formulates the property in a manner that is appropriate only when the symmetry method is not used. Because the symmetry mapping in Figure 7 always rotates the system such that server 1 has the token, only client 1 ever gets to the critical section in the symmetry-reduced state space. However, the rotation does not prevent client 0 from trying to go to the critical section. What happens is that just when client 0 is about to get to the critical section, it becomes client 1. So ASSET incorrectly reports that client 0 tried to go to the critical section but never got there.

To solve this problem, the state variable `c0now` was added to the model. It keeps track of the current number of the original client 0. The verification results became correct, but the reduction in the size of the state space was lost entirely. This is a problem of not just ASSET, but the symmetry method in general.

The counterexamples printed by ASSET may contain symmetry hops. However, in the experience of the present author, they have not harmed the interpretation of counterexamples. They may even be helpful. When a counterexample contains many symmetric copies of the same theme, symmetry hops may reduce them to a single copy.

Acknowledgement. I thank the anonymous reviewers for helpful comments.

References

1. Brookes, S. D. & Hoare, C. A. R. & Roscoe, A. W.: A Theory of Communicating Sequential Processes. *Journal of the ACM* 31, 3 (1984) 560–599
2. Clarke, E. M. & Filkorn, T. & Jha, S.: Exploiting Symmetry in Temporal Logic Model Checking. In: Courcoubetis, C. (ed.) *Computer-Aided Verification '93*, Lecture Notes in Computer Science 697 (1993) 450–462
3. Emerson, E. A. & Sistla, A. P.: Symmetry and Model Checking. In: Courcoubetis, C. (ed.) *Computer-Aided Verification '93*, Lecture Notes in Computer Science 697 (1993) 463–477
4. Holzmann, G. J.: *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley (2003) 596 p
5. Jensen, K.: *Coloured Petri Nets, Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science, Springer (1995) 174 p
6. Manna, Z. & Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems – Specification.* Springer-Verlag (1992) 427 p
7. Roscoe, A. W.: *Understanding Concurrent Systems.* Springer, Heidelberg, Germany (2010) 533 p
8. Valmari, A.: *Error Detection by Reduced Reachability Graph Generation.* In: Proceedings of the 9th European Workshop on Application and Theory of Petri Nets (1988) 95–122
9. Valmari, A.: *The State Explosion Problem.* In: Reisig, W. & Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models, Lecture Notes in Computer Science 1491 (1998) 429–528

10. Valmari, A.: *Stop It, and Be Stubborn!* In: Haar, S. & Meyer, R. (eds.) 15th International Conference on Application of Concurrency to System Design, IEEE Computer Society (2015) 10–19, DOI 10.1109/ACSD.2015.14
11. Valmari, A. & Setälä, M.: Visual Verification of Safety and Liveness. In: Gaudel, M.-C. & Woodcock, J. (eds.) *Formal Methods Europe '96: Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science 1051 (1996) 228–247

n		states	edges	time		states	edges	time
2	P	68	140	0.0	S	44	60	0.0
	Y	34	70	0.0	B	22	30	0.0
3	P	468	1 350	0.0	S	219	327	0.0
	Y	156	450	0.0	B	73	109	0.0
4	P	2 928	10 880	0.0	S	920	1 432	0.0
	Y	732	2 720	0.0	B	230	358	0.0
5	P	17 280	78 600	0.1	S	3 505	5 625	0.0
	Y	3 456	15 720	0.0	B	701	1 125	0.0
6	P	98 064	527 760	0.2	S	12 540	20 772	0.1
	Y	16 344	87 960	0.1	B	2 090	3 462	0.0
7	P	541 296	3 364 200	0.8	S	43 015	73 899	0.2
	Y	77 328	480 600	0.4	B	6 145	10 557	0.1
8	P	2 927 232	20 632 320	4.5	S	143 408	256 880	0.4
	Y	365 904	2 579 040	1.6	B	17 926	32 110	0.2
9	P	15 583 104	122 821 920	30.0	S	469 053	879 885	1.4
	Y	1 731 456	13 646 880	10.0	B	52 117	97 765	0.3
10	P	81 933 120	714 052 800	262	S	1 514 900	2 984 860	4.6
	Y	8 193 312	71 405 280	59.5	B	151 490	298 486	0.9
11	P	–	–	341	S	4 852 771	10 057 839	16.3
	Y	38 771 136	370 202 400	339	B	441 161	914 349	2.6
12	P				S	15 464 040	33 719 400	60.1
	Y	–	–	1039	B	1 288 670	2 809 950	9.1
13	P				S	..	..	65.0
	Y				B	3 777 949	8 659 221	30.0
14	P				S			
	Y				B	11 116 762	26 741 542	96.1
15	P				S			
	Y				B	32 826 001	82 708 765	353
16	P				S			
	Y				B	..	..	131

Table 1. Results on the demand-driven token ring. "P" = plain, "S" = stubborn sets, "Y" = symmetries, and "B" = both. "–" denotes that 10^8 states was exceeded. ".." indicates memory overflow. The times are in seconds and do not include the compilation. The experiments were run on a year 2009 laptop with a 1.6 GHz dual core processor and 1.954 GiB (that is, 2.098 GB) of RAM.