A State Space Tool for Models Expressed in C++
(tool paper)

Antti Valmari
Department of Mathematics
Tampere University of Technology
Tampere, Finland
Email: 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. Its name is ASSET. Model checking takes place by compiling the model and the tool together, and executing the result. This approach facilitates very fast execution of the transitions of the model. 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.

Index Terms—model checking; stubborn sets; symmetries; safety; progress

I. INTRODUCTION

This publication discusses A State Space Exploration Tool called ASSET. It is written in C++. The model under verification is represented as a piece of C++ code that uses the pre-defined type state_var 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 includes the former. So the model is compiled into machine code instead of being simulated by ASSET.

This approach leads to very fast execution of the transitions of the model. It also gives great flexibility, because most features of C++ are available for writing the model. On the other hand, it sometimes leads to unnatural-looking models. Furthermore, as such, ASSET cannot perform any syntactic analysis on the model.

Some advanced methods such as stubborn sets and symmetries require such analysis. The modeller may perform the necessary analysis manually and represent the result as certain C++ functions. Alternatively, there could be a preprocessor tool that inputs user-friendly modelling language. The preprocessor reads the model, analyses it at the syntactic level, and writes it and the necessary additional functions in the form suitable for ASSET. The present author hopes that researchers will find ASSET useful and implement preprocessors from their favourite languages. ASSET is available free of charge for scientific and academic use at http://www.cs.tut.fi/~ava/ASSET/.

Throughout this publication, a demand-driven token ring is used as an example. It consists of $n$ customers and $n$ servers. The customers may request and be granted access to a critical section, return from it to the initial local state, and, for reasons discussed in [4], terminate for good. One token circulates in the system. Only the server that has the token may grant access to its customer. To avoid unnecessary work, the token is not circulated when no customer is requesting access. When necessary, wait information is propagated in the opposite direction to the token.

The model of the example system is introduced in Section II. Section III discusses the features that ASSET offers for specifying correctness properties. Sections IV and V 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 I. The times are in seconds and do not include the compilation.

II. THE DEMAND-DRIVEN TOKEN RING

Figure 1 shows the model of the example system. The #ifdef size_par structure makes it possible to specify the number of customers and servers via an option that is given to the C++ compiler. Also many features of ASSET can be controlled in a similar manner. For instance, with the Gnu C++ compiler, the options -Dstubborn -Dsize_par=13 -Dstop_cnt=10000000 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 customer is 2 (the critical section) but that two bits are used for representing the local state of each customer. 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 initial value is 0. The default value of $b$ is 8.

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. 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 64-bit unsigned integers,
#ifdef size_par
const unsigned n = size_par; // number of customers and servers from compilation command
#else
const unsigned n = 6; // default number of customers and servers
#endif

state_var
C[n] = 2, // state of customer 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 customer
T[n] = 1; // true <=> server i has token

#ifdef symm_must
state_var c0now; // the current index of the original customer 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';
}

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] ){  // C[i] = 0, i = 1
      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) }
      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;
    }
  }
  /* Customers */
  #undef goto
  #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. 1. Model of the demand-driven token ring.
The function fire_transition giving the token to customer 1. The construction of the state space. In Figure 1, it is used for implementing whatever operations are necessary before starting.

If its customer has requested access, it moves the customer to transitions.

Each server of the example system has one transition. In

The function nr_transitions 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.

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

The function nr_transitions may also be used for implementing whatever operations are necessary before starting the construction of the state space. In Figure 1 it is used for giving the token to customer 1.

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 changes the state according to the occurrence of t. If t is disabled, then fire_transition must not change 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 1 introduces two versions of a goto(x) macro. They model the server and the customer going to local state x and indicate that the transition was enabled.

| n  | plain states | edges | time | stubborn sets states | edges | time | symmetries states | edges | time | both states | edges | time |
|----|--------------|-------|------|-----------------------|-------|------|-------------------|-------|------|-------------|-------|------|
| 2  | 68           | 140   | 0.0  | 44                    | 60    | 0.0  | 34                | 70    | 0.0  | 44          | 60    | 0.0  |
| 3  | 468          | 1350  | 0.0  | 219                   | 327   | 0.0  | 156               | 450   | 0.0  | 156         | 450   | 0.0  |
| 4  | 2928         | 10880 | 0.0  | 920                   | 1432  | 0.0  | 732               | 2720  | 0.0  | 732         | 2720  | 0.0  |
| 5  | 17280        | 78600 | 0.1  | 3505                  | 5625  | 0.0  | 3456              | 15720 | 0.0  | 3456        | 15720 | 0.0  |
| 6  | 98064        | 527760| 0.2  | 12540                 | 20772 | 0.1  | 16344             | 87960 | 0.1  | 16344       | 87960 | 0.1  |
| 7  | 541296       | 3364200| 0.8 | 43015                | 73899 | 0.2  | 77328             | 480600| 0.4 | 77328       | 480600| 0.4 |
| 8  | 2927232      | 20632200| 4.5| 143408               | 256880| 0.4 | 365904            | 2579040| 1.6| 365904       | 2579040| 1.6 |
| 9  | 15583104     | 122821920| 30.0| 469053               | 879885| 1.4 | 1731456          | 13646880| 10.0| 1731456      | 13646880| 10.0 |
| 10 | 81933120     | 714052800| 262| 1514900              | 2984860| 4.6| 1819312          | 71405280| 59.5| 1519400      | 2984860| 4.6 |
| 11 | –            | –     | 341 | 4852771              | 10057839| 16.3| 38771136         | 37020400| 339| 441161       | 914349| 2.6 |
| 12 | 15464040     | 33719400| 60.1| –                    | –     | 1039| 1288670          | 2809950| 9.1 | 1288670      | 2809950| 9.1 |
| 13 | –            | –     | –   | 7777949              | 8659221| 30.0| 11116762         | 26741542| 96.1| 11116762     | 26741542| 96.1 |
| 14 | –            | –     | –   | 32826001             | 82708765| 353| 32826001         | 82708765| 353 | 32826001     | 82708765| 353 |

TABLE I

RESULTS ON THE DEMAND-DRIVEN TOKEN RING. “–” DENOTES THAT 10⁷ STATES WAS EXCEEDED. “..” INDICATES MEMORY OVERFLOW.
/* 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( C[i] == 2 ){ ++cnt; }
    }
    if( cnt >= 2 ){ return "Mutual exclusion violated"; }
    return 0;
}

/* Check that every customer has stopped. */
#define chk_deadlock

const char*check_deadlock(){
    for( unsigned i = 0; i < n; ++i ){
        if( C[i] != 3 ){ return "Customer not terminated"; }
    }
    return 0;
}

/* Check that the original customer 0 eventually gets access if it wants to. */
//#define chk_must_progress

bool is_must_progress(){ return C[c0now] != 1; }

Fig. 2. The check functions of the model of the demand-driven token ring.

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.

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 same transitions again in them as it fired in the first stage. 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 [4] 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 I.

Outside Table I the is_must_progress function in Figure 2 was switched on in some experiments. ASSET found no errors in the model in Figure 1. 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 customer does not get access. In it, another customer leaves the critical section, requests for access again, and gets access again.

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 AG EF 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 1 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. This explains the “...” entries in Table I.

IV. THE STUBBORN SET METHOD IN ASSET

The implementation of the stubborn set method in ASSET is discussed extensively in [4]. Therefore, it is discussed here only briefly.

Only the basic strong stubborn set method is implemented. However, theorems in [4] tell that if the model is AG EF 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 AG EF 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
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)] ) )
                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;
            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;
    }
}

if( i >= 2*n )
{
    i -= 2*n;
    switch( S[i] )
    {
        case 0: if( C[i] == 1 || ( S[next(i)] == 1 && !T[next(i)] ) )
                stb(i, next(i)+2*n); return;
        case 1: if( !T[i] )
                stb(prev(i)+2*n); return;
            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. 3. The stubborn set obligation rules of the model of the demand-driven token ring.

this transition is in the stubborn set, then also these transitions
must be”. Figure 3 shows the rules used in the experiments
reported in Table I.

The present author did not at first realize the necessity of
the part && !T[next(i)] in case 0 in Figure 1. 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 AG EF 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, customer 0 visits the critical section and both cus-
tomers terminate. Because waiting information may be propa-
gated from server next(i) to server i even if the former has the
token, unnecessary waiting information enters the ring. Even-
tually 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 length of the
counterexample has not been minimized.

The stubborn set implementation in ASSET does not guar-
anteed 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 III ASSET did find the
error with stubborn sets switched on. With n = 8, there were
2472336 states and 17539200 edges without and 163264
states and 293984 edges with stubborn sets.

V. 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, includ-
including [1]–[3].

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.

The modeller or preprocessor must provide the function
symmetry_representative. It must map 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.
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 ){ T[j] = A[j]; }
    #ifdef symm_must
    c0now = (c0now + n-i) % n;
    #endif
}

Fig. 4. The symmetry representative function of the demand-driven token ring.

If the symmetry method has been switched on, ASSET calls 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 swaps, 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 4 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 check_state and check_deadlock analyse a single state, it is not difficult to make them give the same reply on symmetric states. The versions in Figure 2 do so.

It is more difficult with progress properties. For instance, consider the eventual access property “if customer 0 wants to go to the critical section, it eventually gets there”. When symm_must is off, the is_must_progress function in Figure 2 formulates the property in a manner that is appropriate only when the symmetry method is not used. Because the symmetry mapping in Figure 4 always rotates the system such that server 1 has the token, only customer 1 ever gets to the critical section in the symmetry-reduced state space. However, the rotation does not prevent customer 0 from trying to go to the critical section. What happens is that just when customer 0 is about to get to the critical section, it becomes customer 1. So ASSET incorrectly reports that customer 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 customer 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 swaps. 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 swaps may reduce them to a single copy.

ACKNOWLEDGEMENT
I thank the anonymous reviewers for helpful comments.

REFERENCES
[1] Exploiting Symmetry in Temporal Logic Model Checking. In: C. Courcoubetis (ed.) Computer-Aided Verification ’93, Lecture Notes in Computer Science 697 (1993) 450–462
[2] E.A. Emerson and A.P. Sistla: Symmetry and Model Checking. In: C. Courcoubetis (ed.) Computer-Aided Verification ’93, Lecture Notes in Computer Science 697 (1993) 463–477
[3] K. Jensen: Coloured Petri Nets, Volume 2, Analysis Methods. Monographs in Theoretical Computer Science, Springer (1995) 174 p
[4] A. Valmari: Stop It, and Be Stubborn! Submitted for publication.