This is a self-archived version of an original article. This version may differ from the original in pagination and typographic details.

Author(s): Valmari, Antti; Lappalainen, Vesa

Title: Modelling Without a Modelling Language

Year: 2018

Version: Accepted version (Final draft)

Copyright: © Springer International Publishing AG, part of Springer Nature 2018

Rights: In Copyright

Rights url: http://rightsstatements.org/page/InC/1.0/?language=en

Please cite the original version:

Valmari, A., & Lappalainen, V. (2018). Modelling Without a Modelling Language. In M. D. M. Gallardo, & P. Merino (Eds.), SPIN 2018 : Model Checking Software (pp. 308-327). Springer. Lecture Notes in Computer Science, 10869. https://doi.org/10.1007/978-3-319-94111-0_18

Modelling Without a Modelling Language

Antti Valmari and Vesa Lappalainen

University of Jyväskylä, FINLAND
antti.valmari@jyu.fi, vesa.t.lappalainen@jyu.fi

Abstract. Developments in computer hardware and programming languages, in this case C++, have made it feasible to write models of concurrent systems under verification in the programming language, instead of some established modelling language such as Promela. While this does not reduce the usefulness of modelling languages, it offers new possibilities that may be advantageous, for instance, when teaching state space ideas to newcomers or when experimenting with new scientific ideas. In earlier work, we were able to express everything else fairly naturally in C++, except the set of transitions. The present study uses C++ lambda functions to represent naturally transitions that consist of a tail state, guard, body, and head state. We discuss two implementations, a simple one and a faster one. We present measurements demonstrating that the loss of performance compared to the earlier approach is not big. Starting to use our approach is easy, because one only needs to have a C++ compiler and download (not install) one C++ file.

Keywords: explicit state spaces; modelling languages; implementation issues

1 Introduction

In this publication, our focus is on concurrency aspects of systems. Therefore, by a *modelling language* we mean a language that has been designed for model checking concurrency aspects. Often, but not always, the model is an abstraction that, for instance, replaces the computation of a checksum by a nondeterministic choice between two values "correct checksum" and "incorrect checksum". A *programming language*, on the other hand, is a language meant for implementing systems. It need not support concurrency. In an implementation, abstractions of the kind mentioned above are not made.

Even when focusing on concurrency aspects, and therefore abstracting away from many details such as the computation of checksums, a model may have to contain some sequential computation. For instance, in a telecommunication protocol, the number of retransmission attempts may be counted and compared against a pre-defined constant maximum value. Therefore, many modelling languages contain at least some machinery for expressing sequential or functional computation.

If the machinery is powerful enough, then it may be possible to express the system, instead of just an abstraction, in the modelling language. There may be

a compiler that can compile the model into one or more executables that can be installed in the hardware components of the system. In this case, the model is actually a program and the modelling language is a programming language. That is, a language may be both a modelling and a programming language.

Because of the computational complexity of model checking, it is often the case (at least with current state of the art) that the system cannot be verified but its abstraction that focuses on concurrency aspects can. Therefore, although the distinction between modelling and programming languages is not sharp, we feel it important to maintain a distinction between modelling and programming. Modelling is the act of writing something for the purpose of model checking, and programming is the act of writing something for implementation purpose.

There are tools that can model check systems expressed in ordinary programming languages, such as Java PathFinder [21], Bandera (another toolset for Java) [2], and CBMC (C/C++ bounded model checker) [1]. When they succeed, they make it possible to forget about the distinction between programming and modelling. Because of the complexity of verification, they often fail.

A well-designed modelling language, such as Promela [6], FDR-CSP [11], or CPN ML [7, 9], has many advantages. It supports some modelling paradigm very well. It facilitates efficient model checking. If it has many users, it serves as a widely known medium for sharing models. If it is powerful and flexible, it can also be used, albeit perhaps clumsily, for model checking tasks that arise from other domains than what it was designed for. For instance, encoding place/transition Petri nets [10] in Promela would be unnatural but certainly possible. The same can be said about the wolf, goat, cabbage and farmer puzzle, or the knight's tour.

On the other hand, learning a modelling language is a non-trivial task, in particular for a newcomer to model checking of concurrent systems. In addition to learning mundane details of the language, such as how loops and **if**-statements are written, the newcomer must grasp the fundamental ideas of concurrent execution and nondeterministic choice between alternative atomic actions, and various concepts for expressing properties that should be checked on the model. These ideas are radically different from everything that many students and software engineers have encountered before. To avoid state explosion, the newcomer must also learn to use so little memory in the model that it seems ridiculous from ordinary programming point of view. Furthermore, installing SPIN, FDR, or CPN Tools is not absolutely trivial.[1]

In the case of Java PathFinder, Bandera, and CBMC, the students need not learn a new low-level syntax. On the other hand, the use of Java or C/C++ as such makes the above-mentioned fundamental ideas of concurrency and nondeterminism somewhat implicit, making it harder to learn them. In many (albeit not all) algorithm textbooks such as [3], algorithms are expressed in pseudocode,

[1] The first author tried to install SPIN to Ubuntu 16.04 LTS according to the instructions at [12]. It failed because of the absence of `yacc` in the system, but succeeded after installing `yacc`. He also tried both of the Linux precompiled executables at http://spinroot.com/spin/Bin/ in vain.

to avoid hiding the essence behind low-level implementation issues. We believe that for the same reason, when introducing model checking to newcomers, it is advantageous to use a notation that brings concurrency and nondeterminism forward.

From the research point of view, to experiment with an idea, it may be necessary to make modifications to the input language or other features of the verification tool in use. SPIN is distributed freely in source form, so making such modifications is possible. However, because SPIN is a big program, making modifications to it is far from trivial.

It is often possible to express the state space construction problem in terms of guarded transitions that act on shared variables. That is, there is a set of variables, called *state variables*, and a set of transitions. A transition consists of a Boolean function on the state variables and of a (possibly complicated) piece of code that makes assignments to the state variables. The value of the former tells whether or not the transition is *enabled*. If the transition is enabled, then the latter may be executed, assigning new values to zero or more state variables, based on the earlier values of the state variables. This approach makes concurrency and nondeterminism explicit.

Petri nets are clearly an instance of this idea. The processes of Promela can be interpreted in these terms by treating the location of control of each process as an extra state variable. On-the-fly process creation goes beyond the basic version of this model. However, it need not be among the first topics that are taught to a newcomer in model checking.

It is the opinion, and to some extent also the experience, of the first author that it is easier for newcomers to learn the fundamentals of state spaces with guarded transitions on state variables, because they are so different from ordinary programming languages that the learner is not misguided by earlier intuition on sequential programs. For instance, it is sometimes hard for a newcomer to accept that a process may choose the second nondeterministic alternative (e.g., timeout) even if also the first (e.g., inputting a message) is enabled. It becomes less hard, if the alternatives do not look like an ordinary **if**-statement with **else** replaced by `::`. Guarded transitions on state variables are also sometimes a very good formalism for experimenting with new model checking algorithms.

Of course, this does not mean that one should reject established modelling languages and switch to guarded transitions on state variables. It only means that sometimes there are valid reasons for using something else than an established modelling language.

The research that led to the present study started in autumn 2014 as an attempt to give students a small quickly written tool with which they could play with state space ideas, without having to install any program or learn any new syntax. The students had strong background in C++ [13] and sequential programming, varied but mostly rather weak background in theoretical aspects of software and computer science, and little background in concurrency. The idea was that students write the guards and assignment parts of guarded transitions in C++ (where the assignment parts are not restricted to just assignments, but

may contain loops, etc.), and the file containing them is `#include`d to a program written by the first author which constructs the state space. The students also write C++ functions that specify some correctness properties, the simplest example being a function that checks a state and either deems it good or returns an error message in the form of a character string chosen by the student.

This use of a programming language is fundamentally different from how Java or C/C++ is used with Java PathFinder, Bandera, or CBMC. The latter aim at model checking implementations. Therefore, they use the semantics of the programming language as such. Our approach aims at expressing and model checking abstractions (in the sense discussed above). The semantics of concurrency and nondetermism are not picked from the native semantics of C++, but defined outside the definition of C++ and implemented as classes and other C++ mechanisms. The main goal was not to build a heavy-duty verification tool. Instead, it was to make it as easy as possible for newcomers to learn essential ideas behind model checking. Even so, the resulting tool is actually fast.

The first version of the tool suffered from serious weaknesses. Most importantly, the global state was represented as a single unsigned integer, forcing the students to represent state variables as bit segments within it. Despite this, the tool was pedagogically successful. Almost all students understood the idea of exhaustive search and became able to model such systems as the knight's tour, and most students succeeded in modelling a non-trivial concurrent system such as a token ring protocol. At the same time, the first author wanted to experiment with a new way [17, 18] of applying stubborn set / partial order methods, to solve the so-called ignoring problem [4, 14] much better than before. The tool proved suitable for this purpose.

To let the modeller use more than the 32 (or 64) bits of a single unsigned integer to represent the global state, a C++ class `state_var` was written that looks like an ordinary variable to the modeller but behind the scenes operates on the data structure that stores the so far constructed states. As a consequence, all but one aspects of the modelling of concurrent systems and their correctness properties as systems of guarded transitions on state variables became simple and intuitive. The remaining problem was that often the transitions had to be modelled as complicated collections of `switch`- and `if`-statements. At that stage, the tool was given the name ASSET (A State Space Exploration Tool) and published [16].

Then it turned out that the complicated `switch`-statements can be avoided by representing the transitions with the aid of C++ lambda functions. The present study focuses on this idea.

Section 2 introduces the example system used in this study. In Section 3, it is modelled for ASSET. Natural modelling of transitions relies on two C++ classes written for this purpose. A straightforward version of them is shown in Section 4, and a faster but more complicated version is discussed in Section 5. Section 6 presents some measurements comparing the two implementations to each other and to a model based on the use of `switch` and `if` statements. There also is a tiny comparison to SPIN. The study is concluded in Section 7.

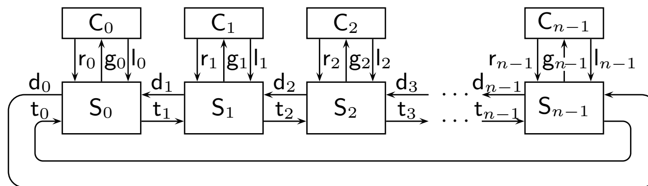

Fig. 1. Overall structure of the demand-driven token ring

2 A Demand-Driven Token Ring

In this section we present the system that is used as the main example in this publication. It is from [15, 16], but we model it in a different fashion.

Fig. 1 shows its architecture. It is a demand-driven token ring consisting of n *clients* $\mathsf{C}_0, \ldots, \mathsf{C}_{n-1}$ and n *servers* $\mathsf{S}_0, \ldots, \mathsf{S}_{n-1}$. There is precisely one *token* in the ring. Each client has a region in its code that is called *critical section.* The purpose of the system is to ensure *mutual exclusion* between the clients, that is, two clients must never be simultaneously in their critical sections. Client i requests for access to its critical section by executing the action r_i. If server i does not have the token, it obtains it as is described below. When it has the token, it grants client i the permission by executing g_i. Now client i is in its critical section. When client i leaves its critical section, it executes l_i to inform the server that it may now give the token to the next server.

When necessary, server i demands the token from the previous server by executing d_i. The demand progresses in the ring until it reaches the server in possession of the token. Let that server be number j. When server j no longer needs the token, it gives it to the next server by executing $\mathsf{t}_{j\oplus 1}$, where $j \oplus 1 = 0$ if $j = n-1$ and $j \oplus 1 = j+1$ if $0 \leq j < n-1$. The token travels in the ring to server i. The servers through which it travels may serve their own clients before passing the token to the next server.

The clients can be easily described precisely. Fig. 2 left shows them as labelled transition systems. Each client has a terminal state (state 3) and a transition to it, to model the fact that a client need not request for access to its critical section if it does not want to. We will now discuss this detail a bit.

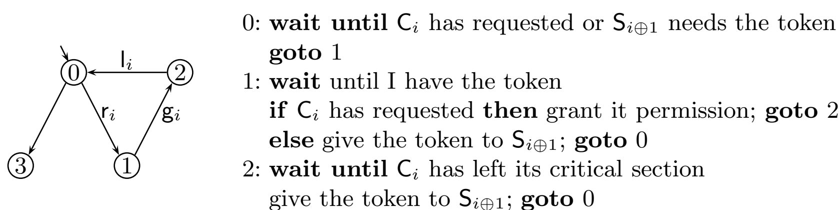

Fig. 2. The clients as labelled transition systems (left) and servers in pseudocode (right)

The termination branch (or some other modelling trick) is necessary to avoid a modelling trap. Assume that after getting the token, each server always waits until its client makes a request, then serves the client, and only then passes the token forward. This is unacceptable, because it may take a long time before the client makes the request, forcing other clients to wait unnecessarily. What is worse, if the server's own client never requests, then other clients that have requested are never served. However, if a client consists of just an r_i-g_i-l_i-cycle, then this error is not caught. This is because then the model lacks the possibility of the server's own client not requesting in the described situation. The request transition is then enabled, and will therefore occur, if nothing else can happen in the system. The termination branch makes it possible for the request transition to not occur, resulting in a deadlock and thus revealing the error.

The typical way of solving the above-mentioned problem is the use of so-called weak fairness assumptions as described in [8]. However, the termination branch is also valid, and has certain advantages. It tends to make non-progress errors manifest themselves as illegal deadlocks instead of unfair cycles, making them technically easy to detect. It is also better compatible with so-called partial order / stubborn set methods that help keep the size of the state space manageable. It has a solid theoretical justification via the notion of stable failures in process algebras. Further discussion can be found in [19] or in Section 3 of [17, 18].

The labelled transition systems that represent the servers are too big and unintuitive to be shown here. (Please see [15] for a slight variant.) This is because the behaviour of a server depends on the presence or absence of a request by its client, the presence or absence of a demand for the token by the next server, whether the server itself has expressed such a demand, and whether it has the token. Fig. 2 right describes the servers via a mixture of a state machine and natural language. In the next section we will see a more precise description in the form of guarded transitions on state variables.

3 An ASSET Model of the Example System

In this section we illustrate that, using C++ lambda functions, our example system can be expressed as a guarded transition model, in a readable fashion.

In many modern programming languages, libraries constitute an intermediate layer between a program and the core language. When a program uses, say, a sorting subroutine that is picked from a library, it gets a significant piece of code that is usually not counted as a part of the program, although it is not part of the core language either. If a better sorting subroutine becomes available, the program may be easily modified to use it.

Similarly, the model in this section relies on some facilities, one version of which will be developed in Section 4 and an alternative version in Section 5. We do not consider these facilities as part of the model proper, since they play a similar role as subroutines picked from a library. (We do admit, however, that we have not yet implemented them in the form of a library, nor made them as generic as they could be.) To verify the model, one only needs to have a C++

compiler, download the program `asset.cc`,[2] copy the model with the supporting facilities to the file `asset.model`, compile `asset.cc`, and run the result.[3]

To model the system for ASSET, we first specify the size of the system. The simplest way to do this would be

```
const unsigned n = 6;
```

However, to make it easier to experiment with systems of different sizes, we exploit C++ macros as follows:

```
#ifdef size_par
  const unsigned n = size_par;
#else
  const unsigned n = 6;
#endif
```

This means that if the compilation command specifies a value for `size_par`, then it is used as the size of the system; and otherwise the size is 6. With the `g++` compiler, the value is given with the option `-Dsize_par=10` (with any natural number in the place of `10`).

Next we introduce the state variables. Each client has four states: 0 = idle, 1 = requested, 2 = critical, and 3 = terminated. There are n clients. We model them with `C[n]`, that is, an array named `C` whose indices run from 0 to $n-1$. The tokens are modelled with an array `T[n]` such that `T[i] == true` if and only if server i has the token. If we let the transitions of the server read the state of the client directly, then the server need not store the piece of information whether the client has requested or not. The same idea can be applied to the demand. These simplify the server so much that three states suffice: 0 = initial, 1 = waiting for the token, 2 = waiting for the client to leave its critical section. These states match Fig. 2 right. The following is written to the model:

```
enum { idle, requested, critical, terminated };
enum { initial, wait_token, wait_client };
state_var C[n], S[n];
state_bit T[n];
```

The only things above that are not readily available in C++ are `state_var` and `state_bit`. They are classes that have been defined in `asset.cc`. From the point of view of the modeller, variables of their types look like ordinary 8-bit and 1-bit unsigned integer variables.

However, as was explained in [16], behind the scenes `state_var` takes care of the memory management needed to store a state into the state space. The class `state_bit` was added to ASSET when writing the present study. It is important to realize that from the point of view of ASSET, variables of these types do not store state information. Instead, all global states are stored in a C++ vector

[2] http://users.jyu.fi/%7eava/ASSET/asset.cc
[3] http://users.jyu.fi/%7eava/ASSET/run

of unsigned integers. (C++ vector is an array with some special services. Most importantly, it can be extended on-the-fly.)

An individual global state occupies some constant number of successive slots of the vector. A variable of these types only contains the information needed to access the value (as seen by the modeller) of the state variable from within the unsigned integers that represent a single global state. When the model uses a state variable, its value (as seen by the modeller) is accessed from one or another global state depending on the value of a variable that contains the index of the current global state. This variable is part of ASSET and not part of the model.

To fire a transition on a global state, ASSET (not always, see below) copies the state (as a sequence of unsigned integers) to a free sequence of slots, makes the latter be the current global state, and asks the model to fire the transition. If the transition is enabled, the model executes its body, potentially modifying the global state. ASSET checks whether the resulting state has been encountered before. If not, it is made an official state of the state space, the sequence of slots is permanently reserved for it, and a new free sequence of slots is acquired by extending the vector. If the model replies that the transition is disabled, ASSET tries the next transition without copying the state, because it has not changed, because the previous transition did not fire. This speeds up the processing of disabled transitions. We will see later that the processing speed of disabled transitions is important.

By default, all state variables hold initially the value 0, also known as `false`. However, there must initially be one token in the ring. Therefore, we write

```
void initialize(){ T[0] = true; }
```

How to write transitions nicely depends on the modelling paradigm and perhaps also on the system. Classes, macros, or other means may have to be defined. In this section we show the transitions of our example system. The first version of the classes and macros that we used is shown in Section 4. To illustrate experimentation with implementation ideas, in Section 5 we describe another, more complicated version of the classes that uses precisely the same representation of the transitions, but speeds up the construction of the state space. We also describe a small modification to the representation of the transitions that yields further speed-up.

The transitions of the clients are copied almost trivially from Fig. 2 left using the names of states. All transitions of the client except the termination transition are joint with the server. However, each of them has a natural direction of the signal, as shown by arrowheads in Fig. 1. Each transition is modelled at the tail of the arrow, that is, at the sender of the signal. Therefore, the transition that moves the client from `requested` to `critical` is modelled as a server transition.

```
client_tr clients[] = {
  client_tr( idle, terminated ),  // termination transition
  client_tr( idle, requested ),   // request access
  client_tr( critical, idle )     // leave critical
};
```

```
server_tr servers[] = {
  server_tr(
    initial,
    GUARD( C[i] == requested || S[ next(i) ] == wait_token ),
    BODY(),
    wait_token
  ),
  server_tr(
    wait_token,
    GUARD( T[i] && C[i] == requested ),
    BODY( C[i] = critical; ),
    wait_client
  ),
  server_tr(
    wait_token,
    GUARD( T[i] && C[i] != requested && S[ next(i) ] == wait_token ),
    BODY( T[i] = false; T[ next(i) ] = true; ),
    initial
  ),
  server_tr(
    wait_client,
    GUARD( C[i] != critical ),
    BODY( T[i] = false; T[ next(i) ] = true; ),
    initial
  )
};
```

Fig. 3. The transitions of the servers

The transitions of the servers are less trivial. We first need a simple helper function that, given the index of a server, yields the index of the next server.

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

The transitions of the servers are shown in Fig. 3. Each of them consists of four components: tail state, guard, body, and head state. A transition is enabled if and only if the control of the server is at the tail state and the guard evaluates to true. When the transition occurs, it executes the body and moves the control of the server to the head state.

The first transition moves the server from `initial` to `wait_token` when there is a reason for that, that is, its own client has requested or the next server needs the token. When the token is available, the second transition moves the client to its critical section, provided that it has requested for access. If it has not requested for access but the next server needs the token, and the current server has it, then the current server gives the token to the next server and returns to its initial state. The fourth transition is enabled when the server is waiting for the client to leave its critical section, and the client has done so. When it occurs, the server gives the token to the next server and returns to its initial state.

After serving its client, the server pushes the token to the next server even if the latter has not demanded. This prevents the system from executing an infinite cycle, where a client requests and its server serves it again and again, while some other client has requested but is never served. In the system as it is in Fig. 3, after a client is served, the token must circulate the ring before the same client may be served again. This guarantees that the other clients will be served if they want.

The verification tool must also be given some properties to check. We first describe mutual exclusion via a feature that makes ASSET check every state that it has constructed. The `#define` switches this feature on. The function counts the number of clients that are in their critical sections, and returns an error message, if and only if that number is at least two.

```
#define chk_state
const char *check_state(){
  unsigned cnt = 0;
  for( unsigned i = 0; i < n; ++i ){ if( C[i] == critical ){ ++cnt; } }
  if( cnt >= 2 ){ return "Mutual exclusion violated"; }
  return 0;
}
```

To see that this function works, we temporarily changed the initialization function so that it puts two tokens to the system: `T[n/2] = T[0] = true;`. As a consequence, ASSET reported `!!! Safety error: Mutual exclusion violated` and printed the sequence of states of an execution that led to the error. ASSET prints each state using a function provided by the modeller. So the modeller has full control on how each state is printed. Because the sequences leading to errors may be long, it is often a good idea to print states so densely that one line suffices. In our experiments we used the following function that encodes the local states of clients and servers as characters.

```
const char Cchr[] = { '-', 'R', 'C', ' ' }, Schr[] = { 'i', 't', 'c' };
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';
}
```

With $n = 6$, the sequence of states mentioned above is as follows. The two tokens are shown as `*`. They are permanently at positions 0 and 3. We have added comments that describe what happened in each transition. "to CS" abbreviates "to its critical section".

```
-i*-i -i -i*-i -i   Initial state
-i*-i -i Ri*-i -i   Client 3 requested.
Ri*-i -i Ri*-i -i   Client 0 requested.
```

`Ri*-i -i Rt*-i -i`	Server 3 moved to `wait_token`.
`Ri*-i -i Cc*-i -i`	Server 3 moved to `wait_client`, taking client 3 to CS.
`Rt*-i -i Cc*-i -i`	Server 0 moved to `wait_token`.
`Cc*-i -i Cc*-i -i`	Server 0 moved to `wait_client`, taking client 0 to CS.

We also used a function that verifies that when the system has terminated, all clients have terminated on purpose instead of being blocked. If the initialization function is changed so that it puts no token to the system, then ASSET reports `!!! Illegal deadlock: Client not terminated` and shows a sequence of states where one by one, all clients move to `requested`.

```
#define chk_deadlock
const char *check_deadlock(){
  for( unsigned i = 0; i < n; ++i ){
    if( C[i] != terminated ){ return "Client not terminated"; }
  }
  return 0;
}
```

We mentioned earlier that after serving its client, the server pushes the token to the next server even if the latter has not demanded. To illustrate that this is important, we temporarily removed the latter `T[i] = false; T[ next(i) ] = true;` and permanently added the following checking function. It verifies that the system has no infinite execution where client 0 stays permanently in `requested`. This fails in the intentionally broken system, causing ASSET to report `!!! Must-type non-progress error` together with a long sequence of states that ends with a cycle where server 5 repeatedly serves client 5 while client 0 is in `requested`. The other clients have terminated and all servers other than 5 are waiting for the token.

```
#define chk_must_progress
bool is_must_progress(){ return C[0] != requested; }
```

An implementation of `client_tr` and `server_tr` will be described in the next section, and a faster, more complicated implementation in Section 5. The code fragments in this section together with either implementation constitute a complete model that ASSET can check. Our claim is that for a person who knows C++ and the basics of state spaces, the model in this section is reasonably easy to follow. It is also reasonably easy to make experiments by making modifications to the model. The classes in the next two sections are more difficult, but they can be re-used in other models.

4 Simple Transition Classes

In this section we describe a simple version of the generic facilities that the model in the previous section uses. The idea is that in the future, there would be a library from which these and other similar facilities could be picked. The next section discusses a more advanced alternative.

```
typedef bool (*guard_type)( unsigned );
typedef void (*body_type)( unsigned );
#define GUARD(x) { [](unsigned i) {return x;} }
#define BODY(x) { [](unsigned i) {x} }

class server_tr{
  unsigned tail, head; guard_type guard; body_type body;
public:
  static unsigned cnt;
  server_tr(
    unsigned tail, guard_type guard, body_type body, unsigned head
  ): tail( tail ), head( head ), guard( guard ), body( body ) { ++cnt; }
  bool operator()( unsigned i ) const {
    if( S[i] != tail || !guard( i ) ){ return false; }
    body( i ); S[i] = head; return true;
  }
};
unsigned server_tr::cnt = 0;
```

Fig. 4. The server transition facilities

The facilities needed by the transitions of the server are shown in Fig. 4. A guard is a function that takes the index of the server and returns a Boolean value, and a body is a function that takes the index of the server and returns nothing. First these two types of functions are given names. Then two macros are shown that facilitate intuitive syntax for the guards and bodies. Each guard and body is a C++ lambda function, that is, a C++ function that has no name. In the definition of a lambda function, `[]` (or something more complicated) appears in the place of the return type and name of the function.

Each `server_tr` object consists of four components: the tail state, the head state, the guard and the body. The execution of a server transition is defined by `bool operator()`. If the current local state of the server is not the same as the tail state of the transition, the execution terminates immediately returning `false`, indicating that the transition is not enabled. In the opposite case (that is, if these two states are the same), the guard is evaluated. If the guard returns `false`, then again the execution terminates returning `false`. In the opposite case, the transition is enabled. Then the body of the transition is executed, the head state is made the current local state of the server, and `true` is returned.

With the aid of `cnt`, the class counts the number of server transitions that are created. (Being an array instead of a vector, `servers` has no `size` operator.) The last line initializes `cnt` to 0. The three lines above `bool operator()` copy the tail and so on from the command that creates a server transition, to the corresponding fields of the object. They also increment `cnt`.

The class `client_tr` is simpler. It only contains `tail`, `head`, `cnt`, and the operations that manipulate them.

```
unsigned nr_server_tr = 0;

unsigned nr_transitions(){
  initialize();
  nr_server_tr = server_tr::cnt * n;
  return nr_server_tr + client_tr::cnt * n;
}

bool fire_transition( unsigned i ){
  if( i < nr_server_tr ){
    return servers[ i % server_tr::cnt ]( i / server_tr::cnt );
  }
  i -= nr_server_tr;
  return clients[ i % client_tr::cnt ]( i / client_tr::cnt );
}
```

Fig. 5. The firing of transitions

We have described how the client and server transitions become stored in the arrays `clients[]` and `servers[]`. We still have to explain how ASSET uses these arrays when constructing the state space. Fig. 5 shows the code that implements this functionality.

Before starting to construct the state space, ASSET calls `nr_transitions()`, to obtain the number of transitions in the model and to perform whatever initialization is needed. The function calls the initialization function discussed in Section 3, computes the total number of transitions as seen by ASSET, and computes the total number of server transitions as seen by ASSET for a reason that will be discussed soon. The number of client transitions that we wrote is in `client_tr::cnt`, and similarly for the server. The total number of transitions that we wrote is the sum of these. However, from the point of view of ASSET, the transitions of each client and server are distinct from the transitions of any other client or server, although we modelled them as parameterized transitions that got the index of the client or server as the parameter. Therefore, for ASSET, both numbers of transitions must be multiplied by the number of servers. Let M denote the total number of transitions as seen by ASSET.

ASSET tries to fire a transition by calling `fire_transition` with the number of the transition as a parameter. If it returns `false`, then ASSET treats the transition as disabled. If it returns `true`, then ASSET assumes that the transition was enabled and has been executed, changing the values of zero or more state variables. If this happens when ASSET is in the state space construction mode, then ASSET behaves like any state space construction tool, that is, checks whether the resulting state has been encountered before, and, if not, stores it in the state space. Then it either copies the original state to a fresh working area and tries the next transition on it or, if all transitions have been tried on the state, chooses the next state for processing.

For fast access, the total number of server transitions, as seen by ASSET, is stored in the global variable `nr_server_tr`. Let that number be denoted with m. Let s denote the number of server transitions that we wrote (that is, $s = 4$). So $m = ns$. Transition numbers from 0 to $m - 1$ correspond to the servers. Using the modulus operator and integer division, `fire_transition` splits the number to a number in the range from 0 to $s - 1$ and another in the range from 0 to $n - 1$. The former is used for picking a transition from the array `servers`, and the latter is given to the picked transition as a parameter. That is, the latter is the index of the server. The transition is executed by invoking the `()` operator, and the returned Boolean value is forwarded to ASSET as the return value of `fire_transition`.

Transition numbers from m to $M - 1$ correspond to the clients. The function `fire_transition` first subtracts m from them and then processes them similarly to the transition numbers that correspond to the servers.

The transition classes developed in this section can be thought of as a middle layer between the model and the ASSET tool. Further classes could be developed for different needs. For instance, there could be a non-parameterized class to model processes that, unlike our clients and servers, exist in only one copy. Writing a class may be non-trivial, but after it has been written, it may be easily usable in many different models. This is analogous to data structure and algorithm libraries.

5 Faster Transition Classes

In the design of the transition classes in the previous section, simplicity was preferred over performance. First, lambda functions are slower than the methods used in ASSET models until now. Before the present study, individual transitions were written as branches of `if` and `switch` statements that direct the control to the right transition on the basis of the number of the transition and the local states of the clients and servers. The statements were inside `fire_transition`. In the present study, the execution of an enabled server transition involves the invocations of two functions whose addresses are picked from an array. This introduces overhead.

Second, from the point of view of ASSET, the model in Sections 3 and 4 contains $7n$ transitions, three for each client and four for each server. Because a server transition is disabled if its tail state is different from the current local state of the server, and because the guards of the second and third transition of the server are in contradiction, at most one of the four transitions of a server can be simultaneously enabled. For a similar reason, at most two transitions of a client can be simultaneously enabled. This means that during state space construction, numerous calls to `fire_transition` are made that yield `false` because of the local state of the client or server. In `switch`-based models, transitions with different tail states may share the transition number, reducing the number of unproductive calls to `fire_transition`.

To obtain information on the magnitude of these phenomena, we implemented four additional models. By Simple we refer to the model developed in the previous sections. Switch3 uses `if` and `switch` statements in the traditional, optimized manner. Also Switch7 uses `if` and `switch` statements, but it uses the same, non-optimal numbering of transitions as Simple. Lambda4 and Lambda3 exploit an improved way of using lambda functions that we will develop in this section. Lambda4 uses precisely the same transition specifications as Simple, but uses only $4n$ transition numbers. In Lambda3, the two transitions that start at `wait_token` have been merged. Like Switch3, it uses $3n$ transition numbers.[4]

The idea is to reduce the number of ASSET transition numbers and eliminate the tests on the tail states of transitions by sharing transition numbers between transitions with different tail states. The transitions (as seen by the modeller) of a server are partitioned to *levels*. The transitions on the same level share an ASSET transition number. Each level contains precisely one transition for each local state of the server, but this transition may be a special transition $\wp$ that is never enabled.

Consider the introduction of a new transition whose tail state is t. Location t on level 0 is checked, then on level 1 and so on, until a location containing $\wp$ is found or the levels are exhausted. In the latter case, a new level is introduced and all its locations are initialized with $\wp$. In both cases, then the new transition is stored on location t on the level.

When ASSET tries to fire transition number k where k is in the range for the server transitions, the index i of the server and the level ℓ are computed using integer division and modulus by the number of the levels. Then the current local state of the server (that is, `S[i]`) is used to pick the right transition on the chosen level. This takes place by using $s\ell + \texttt{S[i]}$ to index an array, where s is the number of local states of the server, that is, $s = 3$. So this is a constant time operation. Next the guard of the transition is evaluated. If it yields `true`, the head state of the transition is assigned as the current local state of the server, and the body of the transition is executed. These two actions are executed in this order to make it possible for the body to override the default head state. This feature is needed in merging the two server transitions whose tail state is `wait_token`.

Transitions of the clients are partitioned to levels in a similar fashion. Before sending an ASSET transition number to the firing function of `client_tr`, the total number of server transitions is subtracted from it, to make the range of numbers as seen by `client_tr` start from 0.

An idea of how complicated this optimized approach is can be obtained from the fact that Simple consists of 158 lines of code (including comments), Lambda4 of 208, and Lambda3 of 205.

The number of levels needed by a process is the maximum number of transitions of the process that may be simultaneously enabled. It can thus be thought of as the degree of nondeterminism of the process. This is not necessarily the same as the maximum outdegree of a local state of the process, because two

---

[4] http://users.jyu.fi/%7eava/ASSET/MWML/simple.cc, switch3.cc, and so on.

Table 1. Running times

n	hash	Simple	Lambda4	Lambda3	Switch7	Switch3	seconds
7	23	1.38	1.27	1.18	1.22	1.00	3.33
7	24	1.37	1.27	1.19	1.24	1.00	3.28
8	23	1.31	1.23	1.19	1.10	1.00	45.7
8	27	1.44	1.34	1.27	1.15	1.00	29.6
9	27	1.26	1.19	1.17	1.11	1.00	355
9	28	1.30	1.21	1.17	1.13	1.00	321

transitions that share their tail state may have mutually exclusive guards. The two server transitions that start at `wait_token` are an example of this.

The degree of nondeterminism is typically small, often 1 or 2. On the other hand, with the simple technique of the previous section, the number of ASSET transition numbers used by a process is the same as the total number of transitions of the process. This number is 3 for our clients and 4 for our servers, but it is often bigger. For instance, the sender of the self-synchronizing alternating bit protocol in [20] has 22 transitions and its degree of nondeterminism is 2. Therefore, one might expect that the levelling technique of this section might yield significant savings with that protocol, but not necessarily with the example of the present study. The next section reports what happened in our measurements.

6 Measurements

Although our main motivation was pedagogical, it is important that the tool is not woefully slow. In this section we demonstrate experimentally that it is actually quite fast. On the other hand, the effect of the improvements in Section 5 over the implementation in Section 4 will turn out not big on our example.

We analysed the five models introduced in the previous sections with $n = 7$, $n = 8$, and $n = 9$, with various hash table sizes. (Hash table is the data structure from which ASSET checks whether a newly constructed state has been encountered before.) For each n, each of the models has the same number of reachable states and edges. These numbers are shown below. An observation that we will refer to in the sequel is that the number of edges is roughly $1.04n$ times the number of states.

n	7	8	9
states	2 939 328	20 155 392	136 048 896
edges	21 500 640	167 588 352	1 267 270 272

Table 1 shows the times it took to construct and analyse the state spaces by ASSET on a machine with 3.60 GHz clock rate and 16 gibibytes of memory. The analysis algorithm tries to fire each transition twice on each reachable state: first to construct the set of reachable states as usual, and then as a part of an algorithm that checks the property specified by `is_must_progress`.

The rightmost column shows the analysis times of Switch3 in seconds, and the five preceding columns show the relation of the analysis time of each model to the analysis time of Switch3. Although we show two decimals, we point out that the information content of the latter decimal is limited, because the analysis times of identical runs on the same machine varied as much as 18 %. So the second decimal is unreliable. The second column shows the base-2 logarithm of the hash table size used in the experiments. For each n, the lower row uses the hash table size that yielded the fastest runs. To obtain information on the effect of the hash table size, the upper row presents the analysis times with a smaller hash table size, which is 23 in most cases but 27 with $n = 9$, because 23 took too much time. The compiler refused to compile when hash was 29.

The total number of transition numbers (as seen by ASSET) is $7n$ for Simple and Switch7, $4n$ for Lambda4 and $3n$ for Lambda3 and Switch3. Indeed, the analysis times of Switch7 are bigger than those of Switch3, and similarly with Simple, Lambda4, and Lambda3. We saw above that of these $7n$, $4n$, or $3n$ transitions, only about $1.04n$ were enabled on the average in each state. So in all models, most calls of `fire_transition` yield `false`. The processing time of such calls is thus important. Also, as one would expect, the `switch`-based models were faster to analyse than the models that use lambda functions.

However, the differences were at most only 44 %. The effect of the hash table size is of similar magnitude. So there is little point in spending human effort on optimizing analysis speed unless the analysis proves too slow. Even then the first thing is to set the hash table size. A good hash table size is at least roughly the number of reachable states, but not so big that the hash table uses too much of the available memory. In the absence of a better idea, the size of the available memory divided by 100 can be used as a rule of thumb. (An obvious idea for improving ASSET would be to make it choose the hash table size.)

During the development of the models, they were tested on a small 8 years old Linux mini-laptop with $n = 8$ and hash = 23. The analysis times were, of course, much bigger, because the computer was of much smaller performance. We feel that they are worth reporting, because there is an interesting difference. The following table shows the analysis times in seconds. They are user times measured with the `time` command, and they include also the roughly 4 seconds that it took to compile the model and `asset.cc`.

n	hash	Simple	Lambda4	Lambda3	Switch7	Switch3	sec
8	23	1.70	1.48	1.33	1.10	1.00	181

On that environment, the overhead of lambda functions is significant. We speculate that the overhead caused by indirect function calls may have become less significant over the years, thanks to developments in hardware (and compilers?). Clearly the measurements with a modern machine shown above do not suggest that lambda functions would be a serious performance problem.

We pointed out in Section 5 that the sender of the self-synchronizing alternating bit protocol has a high ratio of the number of transitions to the degree of nondeterminism. So we expected the levelling technique to yield much more dramatic improvement than it did in the demand-driven token ring example.

We tested two different versions of the protocol on the old slow laptop. The improvement was small in both cases. It turned out that the manipulation of the fifo-channels of the protocol was so time consuming that it dominated the analysis time. Each time when a message is added to a fifo, the first empty location must be found to put it there, and each time when a message is removed, all contents of the fifo must be moved one step forward. Outside state space methods, the fifo could be made faster by implementing it as a ring buffer. However, in a verification model, such an implementation would cause state explosion by giving the same actual content many different representations.

We tried our approach also on the dining philosophers' model at [5]. Translation from Promela along the lines of Section 3 was straightforward. The framework of Section 4 was used, after removing everything related to clients as unnecessary. We tried up to 14 philosophers, with SPIN max search depth set to 18 000 000 (17 000 000 did not suffice) and `no_show_cnt` switched on in ASSET. ASSET always constructed one state and two edges less than SPIN. When $n \geq 12$, its running time was less than half of that of SPIN. When $n < 12$, ASSET terminated within one second. The machine was a modern laptop.

7 Conclusions

We illustrated how lambda functions can be used to write fairly natural and readable models of systems as guarded transitions on shared variables, with tail and head states. Lambda functions were added to C++ in the 2011 standard, so they are somewhat recent. Because in our application, the use of lambda functions involves calling functions picked from arrays, we expected it to add significant overhead. This proved to be so on an 8 year old mini-laptop, but not on modern machines. (We reported measurements on one modern machine but had tried also three others.) In our measurements, the overhead was so small that there is no point in spending human effort to avoid lambda functions unless the analysis speed has to be optimized to the extreme. Although we did not report them, we also made some experiments with virtual functions. Again, the result was that there is no strong need to avoid them.

We first presented simple classes that made it possible to write natural models using lambda functions. Then we developed more complicated, faster classes. However, in our experiments, the motivation for the faster classes was reduced by the fact that on modern machines, already the simple classes performed not much worse than highly optimized models relying on `switch` and `if` statements.

Our work is initial in that our classes are not universal. Instead, they were designed according to the needs of our example system. However, especially the simple classes are straightforward and can thus be mimicked as needed when modelling other systems. We also successfully re-used a class in another model. The work is initial also in that only three systems were modelled and experimented with. Because of the strong expressive power of C++, we are convinced that many further systems can be modelled, and more universal classes than ours can be developed.

References

1. Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
2. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*, pages 439–448. ACM, 2000.
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.
4. Sami Evangelista and Christophe Pajault. Solving the ignoring problem for partial order reduction. *STTT*, 12(2):155–170, 2010.
5. Edmond Ó Floinn. Model of dining philosophers' problem in the Promela verification language, 2016. https://github.com/oflynned/DiningPhilosophersPromela [Online; accessed 2-May-2018].
6. Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.
7. Kurt Jensen and Lars Michael Kristensen. Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems. *Commun. ACM*, 58(6):61–70, 2015.
8. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification.* Springer, 1992.
9. Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML.* MIT Press, 1990.
10. Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science.* Springer, 1985.
11. A. W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2010.
12. spinroot.com. *SPIN Readme.* http://spinroot.com/spin/Man/README.html [Online; accessed 1-May-2018].
13. Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997.
14. Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.
15. Antti Valmari. Composition and abstraction. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000*, volume 2067 of *Lecture Notes in Computer Science*, pages 58–98. Springer, 2000.
16. Antti Valmari. A state space tool for concurrent system models expressed in C++. In Jyrki Nummenmaa, Outi Sievi-Korte, and Erkki Mäkinen, editors, *Proceedings of the 14th Symposium on Programming Languages and Software Tools*

(SPLST'15), Tampere, Finland, October 9-10, 2015, volume 1525 of *CEUR Workshop Proceedings*, pages 91–105. CEUR-WS.org, 2015.
17. Antti Valmari. Stop it, and be stubborn! In *15th International Conference on Application of Concurrency to System Design, ACSD 2015, Brussels, Belgium, June 21-26, 2015*, pages 10–19. IEEE Computer Society, 2015.
18. Antti Valmari. Stop it, and be stubborn! *ACM Trans. Embedded Comput. Syst.*, 16(2):46:1–46:26, 2017.
19. Antti Valmari and Manu Setälä. Visual verification of safety and liveness. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18-22, 1996, Proceedings*, volume 1051 of *Lecture Notes in Computer Science*, pages 228–247. Springer, 1996.
20. Antti Valmari and Walter Vogler. Fair testing and stubborn sets. *STTT*, 2017.
21. Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.