AsmL Specification and Verification of Lamport's Bakery Algorithm

Matko Botinčan
Department of Mathematics, University of Zagreb, Croatia

We present a specification of Lamport's Bakery algorithm written in AsmL specification language. By exploration of the state space of the induced labeled transition system we show how to verify important safety and liveness properties of the algorithm.

Keywords: AsmL, abstract state machines, formal verification, concurrent programming.

1. Introduction

Bakery algorithm, given by Lamport ([10]) in 1974., is one of the most remarkable solutions for Dijkstra's problem of mutual exclusion in concurrent programming. Although clear in its idea, there is still a necessity for a precise mathematical proof of its correctness. Although Lamport proved the correctness of the algorithm in his original paper ([10]) and even gave another, more concise and nonassertional proof few years later ([11]), other proofs, essentially different in its manner, have appeared in the literature since then. The one which will be central to this paper is based on the construction of an abstract state machine capturing the core of the algorithm on its natural abstraction level ([3]).

1.1. Abstract State Machines

Abstract State Machines (ASMs), formerly known as Evolving Algebras (EAs), were introduced by Yuri Gurevich ([5]) as an attempt to fill the gap between known formal models of computation and practical specification methods. It is strongly believed that it is today's most general and all-inclusive formalism. Even more, there exist theses which prove that any sequential algorithm ([6]) or parallel algorithm ([1]) can be modeled at its natural level of abstraction by an appropriate ASM.

An analogous thesis for distributed algorithms is still a work in progress, while its affirmativeness has been widely evidenced in many practical situations. There exist plentiful examples in the literature where distributed ASMs, as a general model of concurrent multi-agent computation, were successfully applied for modeling various distributed algorithms ([2]).

One convincing example is presented in [3] where a simple and concise proof of correctness of Bakery algorithm in terms of distributed ASMs was given. That article will serve as a basis for writing AsmL specification of Bakery Algorithm and its further analysis in this paper.

1.2. AsmL

AsmL (Abstract State Machine Language) is an executable specification language based on the theory of ASMs ([12]). Its purpose is to provide a framework for creation of easy-to-read executable models of a system that can be defined on arbitrary levels of abstraction. Semantic of such AsmL models is perfectly unambiguous as they have mathematically precise meaning given by corresponding ASMs ([8]).

AsmL specifications are very attractive from the practical point of view as they are executable (it is possible to run them as a program in order to e.g. verify run-time behavior of the specification or to test the conformance of an implementation to the specification) and are written in a literate programming style, so they are easy to comprehend.

Unfortunately, due to limitations on the size of this paper, it is not possible to give a reasonable introduction to the theory of ASMs and a thorough description of AsmL. Fortunatelly, on its basic level, AsmL is easy to comprehend, thus most of the language constructs will be self-explanatory and will not need additional explanation.

The rest of the paper is organized as follows. In Section 2, a description of Bakery Algorithm is given. Its AsmL specification is presented in Section 3. Section 4 is dedicated to verification issues of the AsmL model based on exploration of model's labeled transition system. The final Section 5 gives concluding remarks.

```
Start:
    $R_i := 1$
    $A[i] := 1$
Doorway:
    for all $j \neq i$ read($A[j]$, $R_j$)
Ticket:
    $R_i := 1 + \max_j A[j]$
    $A[i] := 1 + \max_j A[j]$
Wait:
    for all $j \neq i$
        repeat
            read($A[j]$, $R_j$)
            until $A[j] = 0$ or $A[j] > A[i]$ or
                ($A[j] = A[i]$ and $j > i$)
Critical Section
Finale:
    $R_i$:=0
```

Fig. 1. Lamport's Bakery Algorithm.

2. Lamport's Bakery Algorithm

Mutual exclusion problem in concurrent programming is described as follows. Let P_1, ..., P_n be processes that want, from time to time, to enter a *critical section* of a program code. One has to design a protocol which will be executed by each P_i before entering the critical section, that will prevent two or more processes to reside in the critical section at the same time.

The key idea of the Bakery algorithm is very simple — as in a bakery shop, processes will be given numbers and whichever process has the lowest number will be allowed to enter the critical section. Each process P_i is provided with a shared register variable R_i and a private array $A[1], \ldots, A[n]$ taking positive integer values. The term "shared" for register R_i denotes that every process can read R_i, but P_i is the only one allowed to write into it. Additionally, it is assumed that every register is initialized with value 0.

The algorithm is divided into six consecutive phases: *start*, *doorway*, *ticket* assignment, *wait* section, *critical section* and *finale*. The task of every phase is depicted in Figure 1.

For a start phase, process P_i declares its interest in accessing the critical section by writing 1 into its register R_i and storing this value in the corresponding private array variable. In the doorway section, P_i copies register contents of all other processes into its private array. Note that for every process, particular read operations in this section can happen in any order, concurrently, all at once, etc.

In the next step, process P_i generates a ticket — a number greater than each element in its array, and writes it into R_i and $A[i]$. Throughout the subsequent wait phase, P_i repeats reading register contents of all other processes into its array, until the condition $A[j] = 0$ or $A[j] > A[i]$ or $A[j] = A[i] \wedge j > i$ is satisfied. The last condition resolves the case when two processes get the same ticket — the process with a smaller index gets the priority. Upon completion of this phase, the process is allowed to enter the critical section. At the end, after leaving the critical section, P_i resets its register R_i to 0.

Once again, note that read operations residing in the doorway and the wait section may be executed in many different ways including all sorts of concurrent executions. This is a primary reason why a manual analysis of the Bakery algorithm can be very difficult.

Formal proof of correctness of the Bakery algorithm and its specification in terms of a distributed ASM has been given by Börger, Gurevich and Rosenzweig in [3] and further analyzed in [7]. As we shall see later, a clever choice of a collection of two types of agents which are to perform concurrent computation steps will allow to faithfully model all relevant aspects of the Bakery algorithm in a reasonably simple way.

3. AsmL Specification

In this section we describe the AsmL model (a model whose specification is written in AsmL) of the Bakery Algoritm. In general, a state of every AsmL model is given by values of all vocabulary symbols that occur in the specification (in the terminology of mathematical logic states are first-order structures). An AsmL model can be seen as a transition system whose transition rules specify modifications from one state of the model to the next. Those modifications are often given in a form of *guarded updates*, i.e. assignment statements which are guarded by an `if` statement and are executed when the given condition holds. A run of an AsmL model is a finite or infinite sequence of states S_0, S_1, ... in which each state S_i $(i > 0)$ is obtained from the previous S_{i-1} by executing all transition rules of the program simultaneously (in parallel) at the state S_{i-1}.

We shall begin by defining all variables that occur in AsmL specification of the Bakery Algorithm, and whose values will represent states of the AsmL model. The specification consists of two types of agents: *customer* agents and *reader* agents. At every computation step, each agent resides in a logical state corresponding to a particular phase of the Bakery algorithm. The union of all logical states that a customer and reader agent can be in is represented through enum type `State` in the specification:

```
enum State
   Start
   Doorway
   Ticket
   Wait
   Check
   CS
   Finale
```

Each *customer* agent represents a process. It is identified by a unique value from the set `Customers` which is initialized to be the set of all customer indices:

```
var Customers as Set of Index =
enum of Index
```

The logical state of each customer is taken to be of the type `CustomerState`:

```
type CustomerState = State where
value in {Start, Doorway, Ticket,
Wait, CS, Finale}
```

Current logical state of each customer agent is stored in the `modeCustomer` map (a map in AsmL is similar to a map in the C++ standard library or to an associative array in Perl, i.e. it is a collection that associates unique keys with values):

```
var modeCustomer as Map of Index
to CustomerState =
{i -> Start | i in Customers}
```

Shared register variables and private arrays belonging to processes are implemented via two maps `R` and `A`, where `R(i)` and `A(i,j)` represent process P_i's register variable R_i and private array member $A[j]$, respectively.

```
var R as Map of Index to Integer =
{i -> 0 | i in Customers}
var A as Map of (Index, Index) to
Integer = {(i, j) -> 0 | i in
Customers, j in Customers}
```

Customer agent (Figure 2.) are represented by the method `Customer(x)`, where `x` is the index of the customer. The code is divided into several steps (each step being guarded by adequate `if` statements) corresponding to the homonymous Bakery Algorithm phases. Steps can be executed only sequentially (remember that AsmL statements are always executed in parallel) as their execution is controlled via the `modeCustomer` map's values of the given customer.

In order to preserve the freedom of choice of an ordering of read operations in the `Doorway` and `Wait` phases, reader agents `Reader(x,y)`, where `x` and `y` are indices of customers, are introduced. During those two phases, each reader agent reads the content of the `y`'s register `R(y)` into `x`'s array component `A(x,y)`. Additionally, reader agents also do the job of verifying the condition which needs to be satisfied prior to entering the critical section.

During the computation, each reader agent can be in one of the logical states represented by the type `ReaderState`:

```
Customer(x as Index)
   if modeCustomer(x) = Start then
      A(x,x) := 1
      R(x) := 1
      modeCustomer(x) := Doorway
      Update()
   if modeCustomer(x) = Doorway and forall y in Customers where y <> x holds
   modeReader(x,y) = Wait then
      modeCustomer(x) := Ticket
   if modeCustomer(x) = Ticket
      let ticket = 1 + (max A(x,y) | y in Customers)
      A(x,x) := ticket
      R(x) := ticket
      modeCustomer(x) := Wait
   if modeCustomer(x) = Wait and forall y in Customers where y <> x holds
   modeReader(x,y) = Doorway then
      modeCustomer(x) := CS
   if modeCustomer(x) = CS then
      modeCustomer(x) := Finale
   if modeCustomer(x) = Finale then
      R(x) := 0
      modeCustomer(x) := Start
```

Fig. 2. Customer agent.

```
type ReaderState = State where
value in {Doorway, Check, Wait}
```

Current logical state of each reader agent is stored in the `modeReader` map:

```
var modeReader as Map of (Index,Index)
to ReaderState = {(i, j) -> Doorway |
i in Customers, j in Customers}
```

The specification of the reader agent is given in Figure 3. As we have selected that customer and reader agents share their logical state identifiers `Doorway` and `Wait` for doorway and wait phases, necessary synchronization between agents is managed very easily. Thus, it is assured that `Doorway` and `Wait` steps in a customer agent `x` can be executed only after all corresponding readers (`Reader(x,y)`, for all indices `y` different from `x`) have executed their `Doorway`/`Wait` and `Check` steps, respectively.

```
Reader(x as Index, y as Index)
   if modeReader(x,y) = modeCustomer(x)
   then
      A(x,y) := R(y)
      if modeReader(x,y) = Doorway then
         modeReader(x,y) := Wait
      if modeReader(x,y) = Wait then
         modeReader(x,y) := Check
   if modeReader(x,y) = Check then
      if A(x,y) = 0 or A(x,y) = 0 or
      A(x,y) > A(x,x) or
      (A(x,y) = A(x,x) and y > x) then
         modeReader(x,y) := Doorway
      else
         modeReader(x,y) := Wait
```

Fig. 3. Reader agent.

4. Exploration of the AsmL Model

Every AsmL model induces a (possibly infinite) labeled transition system (LTS), where a LTS is defined as an ordered quadruple (s_0, S, L, R) in which:

- — s_0 is the initial state defined by the initial values of all variables in the AsmL model;
- — S is the set of all reachable states (reachable from the initial state of the model through invocation of methods);
- — L is the set of all invocations (methods with actual arguments);
- — $R \subseteq S \times L \times S$ is the smallest transition relation that contains all (s, a, t) such that the invocation a can be executed in the source state s and executing a in the state s yields the target state t.

A run of an AsmL model consists of a finite or infinite path through states of the induced LTS. Thus, verifying desired properties of an AsmL model reduces to verifying corresponding properties of states on a path in the induced LTS.

In the LTS of the AsmL model of the Bakery algorithm each state will consist of values of all maps from the model: `modeCustomer`, `modeReader`, `R` and `A`. In the initial state, the value of all keys in `modeCustomer` is `Start`, in `modeReader` is `Doorway`, while the initial values of all keys in `R` and `A` are 0. Further, the set of all invocations in the LTS consists of the following invocations:

— `Customer(x)` where the parameter `x` takes values from the set `Customers`;

— `Reader(x, y)` where the parameters `x` and `y` take values from the set {`(x,y) | x in Customers, y in Customers where x <> y`}

Note that the LTS will have infinitely many states, as ticket values assigned in the doorway phase (depending on the number of competing processes) can be arbitrary large. One can create a state space of manageable (finite) size by introducing a state filter that will prohibit adding a new state to the LTS in which the assigned ticket number is greater than some predefined constant.

Another approach for the state space reduction would be to execute certain kind of update for each process prior to entering the doorway phase which will artificially lower down contents of all registers and arrays by allowed amount. This idea is presented in the following `Update()` method:

```
Update()
   if exists x in Customers, y in
   Customers where A(x,y) > 0 then
      var m = min A(x,y) | x in
      Customers, y in Customers
      where A(x,y) > 0
      if m > 1 then
         forall x in Customers, y in
         Customers where A(x,y) > 0
            A(x,y) := A(x,y) - (m-1)
         forall x in Customers
         where R(x) > 0
            R(x) := R(x) - (m-1)
```

Note that the `Update()` method is just a helper function and will not be exposed as an invocation in the LTS (it is to be executed only internally by customer agents).

The fundamental safety property that has to be verified in order to establish correctness of the Bakery algorithm is that no two processes are allowed to access their critical sections at the same time. In the context of the associated AsmL model, this means that there must not exist a state where the following predicate is satisfied:

```
exists i in Customers, j in Customers
where i <> j and
modeCustomer(i) = State.CS and
modeCustomer(j) = State.CS
```

This can be used as a halting condition during the generation of the LTS (in which case the procedure actually amounts to model-checking of the given formula on the LTS).

Example of an important liveness property for a mutual exclusion protocol such as the Bakery algorithm is absence of a dead-lock. If we mark as accepting states all states in the LTS for which the condition

```
forall i in Customers holds
modeCustomer(i) = State.Start
```

holds, checking for absence of a dead-lock reduces to verifying whether some accepting state is reachable from every LTS state.

The described procedures can be performed automatically by appropriate software tools. In

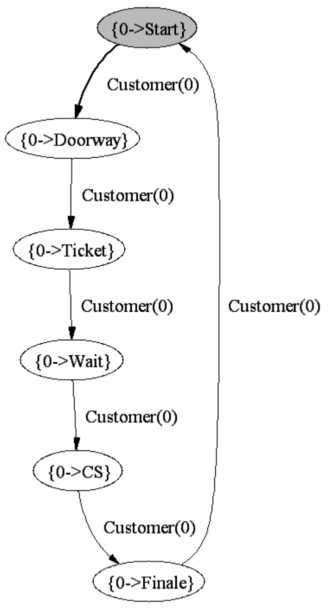

Fig. 4. LTS for a single process (states grouped by modeCustomer).

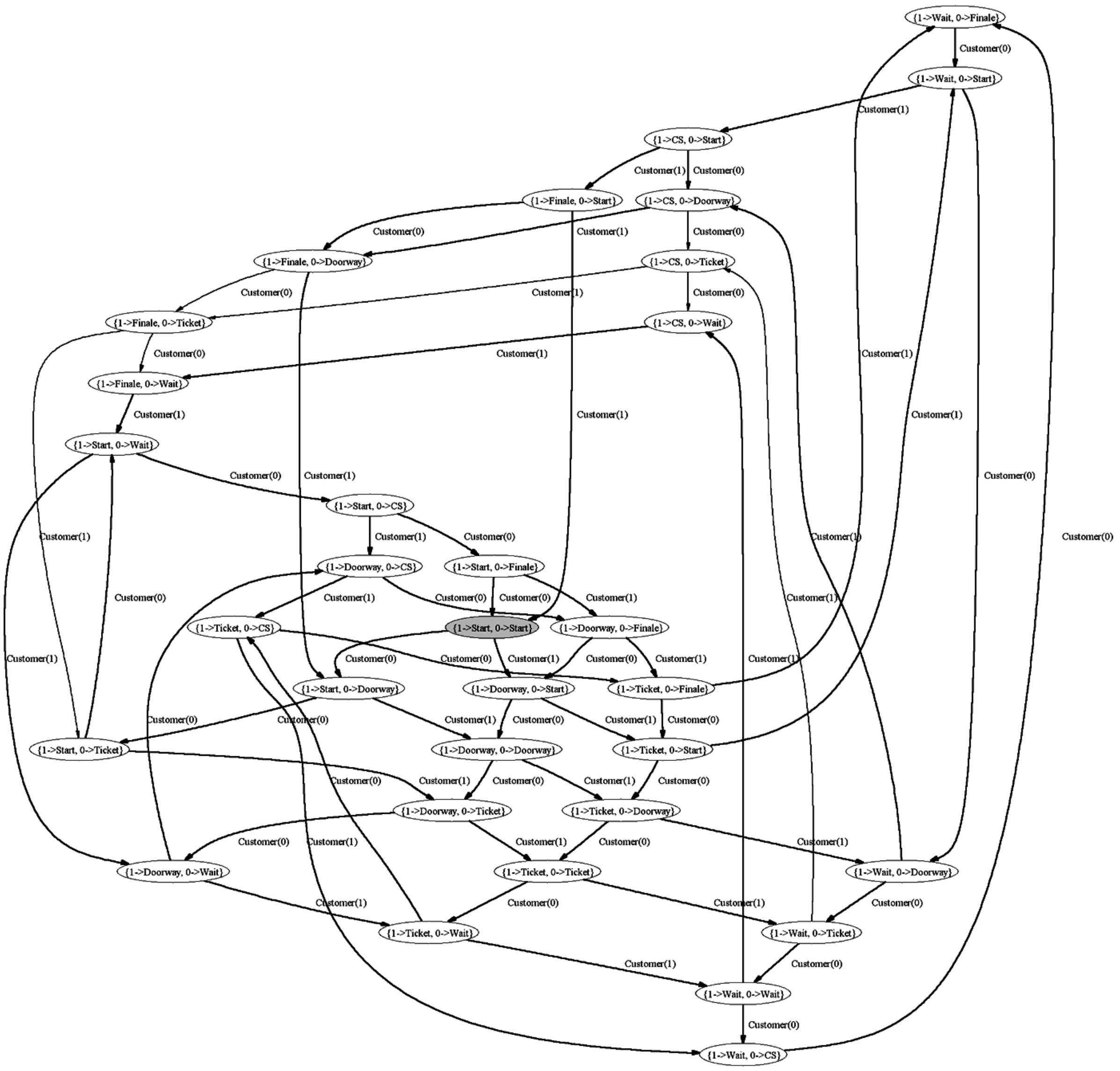

Fig. 5. LTS for two processes (states grouped by `modeCustomer`).

this work we have used SpecExplorer, a model-based testing tool developed by the Foundations of Software Engineering group from Microsoft Research ([4]), but other explicit-state model-checkers could be used for this purpose, too.

Illustrative examples for the case of one and two processes are shown in Figures 4 and 5. The states in both LTS-s are grouped by the values of the map `modeCustomer`, producing in that way a finite directed graph with nodes labeled with corresponding values of the map and edges labeled with invocations.

It should be noted that properties for only fixed number of processes can be proved in this way. Full proof would still need an additional statement that shows if a property holds for n processes, then it holds for $n+1$, too. Nevertheless, the generated LTS-s can be used to understand and validate the overall behaviour of the system in much more natural way than it is possible to do with handwritten proofs.

5. Conclusions and Future Work

In this paper, we have presented an AsmL specification of Lamport's Bakery algorithm. Our primary goal was to illustrate how to verify important safety and liveness properties of the algorithm by exploration of the state space of the induced LTS of the given AsmL model.

This process would naturally proceed with testing of the AsmL specification against an implementation of the Bakery Algorithm via appropriate wrapper functions (i.e. to do the conformance testing). This kind of model-based testing can be done automatically with the aforementioned SpecExplorer software tool.

Additionally, it would also be interesting to consider an instance-based model of the Bakery Algorithm. Besides exposing the strong side of AsmL to deal with object-oriented models, a more transparent model of the Bakery Algorithm with more informative classification of actions could be obtained in this way.

6. Acknowledgment

The author would like to thank Dean Rosenzweig for constructive remarks and helpful comments on the first version of this paper.

References

[1] A. BLASS, Y. GUREVICH, Abstract State Machines Capture Parallel Algorithms. *ACM Transactions on Computation Logic*, 2003; 4(4): pp. 578–651.

[2] E. BÖRGER, Abstract State Machines: *A Method for High-Level System Design and Analysis.* Berlin: Springer-Verlag, 2003.

[3] E. BÖRGER, D. ROSENZWEIG, Y. GUREVICH, The Bakery Algorithm: Yet Another Specification and Verification. In: E. Börger, editor. *Specification and Validation Methods.* Oxford University Press, 1995, pp. 231–243.

[4] Foundations of Software Engineering web page: `http://research.microsoft.com/fse`.

[5] Y. GUREVICH, Evolving Algebras 1993: Lipari Guide. In: E. Börger, editor. *Specification and Validation Methods.* Oxford University Press, 1995, pp. 9–36.

[6] Y. GUREVICH, Sequential Abstract State Machines capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 2000; 1(1): pp. 77–111.

[7] Y. GUREVICH, D. ROSENZWEIG, Partially Ordered Runs: a Case Study. In: Eds. Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors. Abstract State Machines: Theory and Applications, *Proceedings of ASM'2000.* Springer LNCS vol. 1912 (2000), pp. 131–150.

[8] Y. GUREVICH, B. ROSSMAN, W. SCHULTE, Semantic Essence of AsmL. *Technical Report* MSR-TR-2004-27; Microsoft Research, 2004.

[9] J. K. HUGGINS, C. WALLACE, An Abstract State Machine Primer. *Technical Report* CS-TR-02-04; Computer Science Department, Michigan Technological University, 2002.

[10] L. LAMPORT, A New Solution of Dijkstra Concurrent Programming Problem. *Communications of the ACM*, 1974; 17(8): pp. 453–455.

[11] L. LAMPORT, A New Approach to Proving the Correctness of Multiprocess Programs. *ACM Transactions on Programming Languages and Systems*, 1979; 1(1): pp. 84–97.

[12] Microsoft Research. AsmL: *The Abstract State Machine Language*, Microsoft Corporation, 2002. `http://research.microsoft.com/fse/asml`.

Recived: June, 2005.
Accepted: October, 2005.

Contact address:
Matko Botinčan
Department of Mathematics, University of Zagreb
Bijenička cesta 30, 10000 Zagreb, Croatia
e-mail: `mabotinc@math.hr`

MATKO BOTINČAN is a Ph.D student at the Department of Mathematics, University of Zagreb, Croatia, where he received B.Sc (2002) and M.Sc (2005) degrees in mathematics. His fields of interest are finite and algorithmic model theory, computer-aided verification and combinatorial optimization.