# WHAT's THE TITLE?

Jonathan Sharyari

30th January 2013

## Abstract

In 1985, Suzuki and Kasami presented a distributed algorithm for mutual exclusion. In this paper (?), the claim of mutual exclusion, deadlock and starvation freedom are tested using model checking techniques. The conclusion...

# 1 Introduction

SOME BULLSHIT

## 1.1 Principles of Model Checking

Model checking is the process of automatically verifying the correctness of a system description with respect to its specifications, by examining a model corresponding to the system. The result of model checking is in general a certificate of correctness, or a counterexample showing why the specifications are not met.

### 1.1.1 The Spin Model Checker

There are many tools for model checking available, with different approaches to model checking and with different strengths and weaknesses. The SPIN Model checker (typically written Spin) is a widely used tool for model checking distributed and concurrent systems. It uses a simple C-like modelling language (Promela) and performs model checking by generating C code, thus not only makes use of the efficient memory and speed optimizations available for C, but also allows for direct insertions of C-code to the model.

SOMETHING ABOUT LTL CLAIMS -> BUCHI AUTOMATA -> BACK.... Read the book you lazy ass so-called science student.

What is mutual exclusion, starvation freedom and so on. (Would that be expecting too little?)

# 2 Suzuki and Kasami's algorithm

The Suzuki/Kasami algorithm (from this point abbreviated SKA), is a distributed mutual exclusion-algorithm. (SOMETHING ABOUT THE NUMBER OF MESSAGES NEEDED, FOR N NODES)

Each of the N computer nodes between which mutual exclusion is to be realized runs two processes P1 and P2. Each node has two local arrays RN and LN and a queue Q, all of which are accessible by P1 and P2.

The array RN holds the latest received request identifier for each other node. Similarly, LN holds the request identifier of the latest carried out request of all the other nodes, of which the node has been notified. Since the algorithm is distributed, the local arrays RN and LN are not the same for each node, and for each node represent only the information of which the node has been informed. The local queue Q holds the nodes that are currently requesting the privilege, in the order the requests have been received (first come, first serve ??).

When a node $node_i$ wants to enter its critical section with its main process P, it needs to have the privilege to do so. In case the node already holds the privilege, it enters the critical section directly, without informing the other processes or updating the arrays RN and LN. Otherwise a REQUEST(i, n) message is sent to all other nodes, where i is the nodes identifier (index) and n is the request identifier.

A requesting node will wait until it receives a PRIVILEGE message, before accessing its critical section. When it is done, it updates its queue by appending all the nodes

requesting, that are not already in the queue. Then, a PRIVILEGE is sent to the node first in Q, and that node is removed from the queue.

In process P2 REQUEST(i,n) messages are received. The array RN is updated, so that the corresponding entry RN[i] is set to n. If the node currently holding the privilege is not itself waiting for the privilege, it will forward the privilege to $node_i$ by sending a PRIVILEGE(Q, LN) message.

The pseudo-code proposed in the original paper is listed in Appendix A.

# 3 Problem formulation

(NOTE THAT SK MUST BE DEFINED BEFORE THIS) The SK algorithm is claimed to guarantee mutual exclusion, starvation freedom and (ANYTHING ELSE?). The purpose of this project is to investigate these claims, using model checking techniques. Also (SOME BULLSHIT HERE).

# 4 Model of SKA in Promela

Promela is a modelling language with a relatively small set of predefined types and function. Because of this, a model of the pseudo code in Appendix A must be implemented using the set of instructions available in promela.

## 4.1 Nodes

In the SKA algorithm, every node has two processes, and these processes have shared variables. This object oriented style requires different levels of variable access, i.e. variables that can be accessed by several processes (meaning it is not local), but not by all processes (meaning it is not global).

Since no shared, but only global and local variables exist in promela, the model uses global variables. The global variables belong to a node, and must only be accessed by processes within that node. To ensure this, every node has a node identifier, and every process is passed this identifier as an argument.

The convention is, that for every shared variable, an array of those variables are created - one for each node. The $i$:th array cell belongs to node $i$, and is the only cell in the array that processes of node $i$ may access.

As an example, consider the boolean variable `requesting` belonging to node $i$, which must be accessible both by function P1(i) and P2(i). In promela, an array `requesting[N]` is created, and the processes may only access the cell `requesting[i]`.

## 4.2 Queue

A queue data structure could be implemented in several ways in promela, naturally suited for different applications. We note that in the SKA algorithm, the key features of the queue is first that values are read *first-in first-out* (FIFO) and second the ability to determine whether a certain value is already in the queue or not.

The promela channel data type is a FIFO data structure, but it provides no means of determining if a value is already in the queue or not. To surmount this problem, a new *Queue* data structure is defined;

```
1  typedef Queue {
2    chan ch = [N] of {short};
3    bool inQ[N];
4  }
```

This implementation draws upon the fact that a queue in this setting, needs at most store N values, since values may only be added if they are not already in the queue, and the range of numbers that are to be stored are the node indexes ranging from 0 to N-1.

To add a new value $n$ to the queue, it must first be checked that the value is already in the queue (`inQ[n]` is false). Then the value is added to the channel `ch`, and `inQ[n]` be set to true.

To remove the top value from the queue, the opposite is done. The value $n$ is read from the channel `ch`, and `inQ[n]` is set to false, indicating that the value $n$ is not in the queue.

### 4.3 Messages

### 4.4 Event Handling

This part should explain why P2 is waiting in a "busy" loop. I.e., such functions are made to avoid busy loops -> loops are not busy in promela -> doesn't matter that we make busy loops.

## 5 Algorithms

### 5.1 General outline

## 6 Results

## 7 Discussion

## 8 references


1. Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin

   CGAL manual chapter 20, 2D Arrangements

   http://www.cgal.org/Manual/3.3/doc_html/cgal_manual/Arrangement_2/Chapter_main.html

2. O. Aichholzer, R. Fabila-Monroy, T. Hackl, M. van Kreveld, A. Pilz, P. Ramos, und B. Vogtenhuber

   Blocking Delaunay Triangulations.

   Proc. Canadian Conference on Computational Geometry, CCCG 2010, Winnipeg, August 9/11, 2010.

## 8.1 images

- Delaunay circumcircles, GNU Free Documenation Licence, Nü es
  http://commons.wikimedia.org/wiki/File:Delaunay_circumcircles.png
- Delaunay triangulation 1-3, public domain, user Capheiden
  http://upload.wikimedia.org/wikipedia/de/1/17/Voronoi-Delaunay.svg
  http://upload.wikimedia.org/wikipedia/de/4/48/Voronoi-Diagramm.svg
  http://upload.wikimedia.org/wikipedia/commons/1/1f/Delaunay-Triangulation.svg

```
1   const I: Integer;  (* the identifier of this node *)
2     var HavePrivilege, Requesting:  bool;
3     j, n:  integer;
4     Q:   queue of integer;
5     RN, LN:  array[l . . N] of integer;
6
7     (* The initial values of the variables are:
8     HavePrivilege = true in node 1, false in all other nodes;
9     Requesting = false;
10    Q = empty;
11    RN[j] = −1, j = 1,2, . . . , N;
12    LN[j] = −1, j = 1,2,. . . , N; *)
13
14    procedure Pl;
15    begin
16      Requesting := true;
17      if not HavePrivilege then
18      begin
19        RN[I] := RN[Z] + 1;
20        for all j in 11, 2, . . . , NJ − {Z) do
21          Send REQUEST(1, RN[I]) to node j;
22        Wait until PRIVILEGE(Q, LN) is received;
23        HavePrivilege := true
24      end,
25
26      Critical Section;
27
28      LN[Z] := RN[Z];
29      for all j in 11, 2, . . . , N) − [I) do
30        if not in(Q, j) and (RN[jJ = LN[j] + 1) then
31          Q := append(Q, j);
32      if Q f empty then
33      begin
34        HavePrivilege := false;
35        Send PRIVILEGE(tail(Q), LN) to node head(Q)
36      end;
37      Requesting := false
38    end,
39
40    procedure P2; (* REQUEST(j,n) is received; P2 is indivisible *)
41    begin
42      RN[j] := max(RN[j],n);
43      if HavePrivilege and not Requesting and (RN[j] = LN[j] + 1) then
44      begin
45        HavePrivilege := false;
46        Send PRIVILEGE(Q, LN) to node j
47      end
48    end,
```

Listing 1: Suzuki and Kazami's algorithm