Simulation and Validation Tool for Self-Stabilizing Protocols

(Abstract) *

Sandeep K. Shukla *Daniel J. Rosenkrantz* *S. S. Ravi*
Department of Computer Science
University at Albany – State University of New York
Albany, NY 12222
Email: {sandeep,djr,ravi}@cs.albany.edu

Abstract

This abstract describes a project on developing a simulation and validation tool for *self-stabilizing* distributed protocols. The tool is based on an existing tool from Bell Laboratories called SPIN [Hol90]. Our tool basically provides a front end that can handle a specific input syntax in which most self-stabilizing algorithms in the literature are presented. This input syntax is similar to Dijkstra's guarded command language with a slightly different semantics. Further more, one distinct feature of this syntax is that it allows users to specify the architecture (e.g. ring of processors, complete graph, linear array etc.) with other relevant informations (uniform vs. nonuniform, oriented vs. unoriented etc.). This provides the designers of self-stabilizing protocols with the advantage that they do not have to translate by hand their protocols into PROMELA which is the input language for SPIN.

We also plan to handle arbitrary architecture in the input language in the future. We also discuss the possible ways to enhancing our tool for validation of the protocols.

Another advantage of using SPIN at the core is that in PROMELA one can model various restricted kinds of transient faults. As a result, in our input language we can provide syntax for specifying various kinds of transient faults and that can be translated into PROMELA by our frontend automatically. This will give researchers facility to experiment with the various kinds of transient faults that their protocols might tolerate. Also one can validate the specifications of the problems via simulation in our tool.

1 Introduction

Design of distributed protocols is a challanging task due to the complex interaction between distributed agents and combinatorial explosion of the state space of the system which renders the system difficult to analyze. Self-Stabilizing distributed protocols are even more difficult to design because of the extra requirements that they are tolerant to transient fault in a benign fashion. Self-Stabilizing protocols can start at any arbitrary system state and converge in a bounded amount of time to a legitimate system

*This research was supported by NSF Grants CCR-90-06396 and CCR-94-06611.

state. As a result, if a transient failure corrupts the system state, the protocols will malfunction for a bounded amount of time but will recover by themselves to a legitimate system state. Researchers have found it very difficult to prove the self-stabilizability of protocols and the correctness of even very simple protocols become extremely difficult to prove formally [Gou96]. It has been the experience of the authors and many other researchers in the field that that very simple self-stabilizing protocols which have apparently correct proof of stabilizability turn out to have subtle bugs. On the other hand, there are simple protocols which are stabilizing but they are immensely difficult to prove to be stabilizing [Gou96]. As a result, a simulation and validation tool for self-stabilizing protocols that will aid in the design of such protocols and help validating them is extremely needed.

This abstract describes an ongoing project for developing a simulation and validation tool for self-stabilizing protocols. The core of the tool is based on an existing protocol simulation and validation tool called SPIN [Hol90]. SPIN can directly simulate protocols described in a protocol description language called PROMELA. However, translating a self-stabilizing protocol (as they are presented in the literature) into a PROMELA description requires manual translation into the details of PROMELA which may be cumbersome and error prone. Moreover, PROMELA does not allow direct architectural specification and arbitrary initialization which are some of the essential features that a simulator for self-stabilizing protocols should have.

We have designed a description language for self-stabilizing protocols in an architecture dependent fashion. For example, if the self-stabilizing token ring algorithm [Dij74] has to run on a bidirectional non-uniform ring of size 10, then we specify that in our language as BNRING(10,P,Q); followed by the definition of the leader process P and definition of process Q which is run on the other processors. We have designed a prototype compiler that translates a description of a protocol in our language and generates a **PROMELA** [Hol90] specification. It also incorporates arbitrary initialization of process states in each compilation.

The advantage of our language are as follows: First, we can specify the architecture on which our protocol is supposed to run where as PROMELA does not allow one to specify the protocols in such an architecture dependent way. Second, we take advantage of the existing simulator and validator (called **SPIN** [Hol93]) for PROMELA without requiring the user to know the details of the PROMELA language. We interface our compiler to SPIN and thus obtain a simulator for protocols specified in our language.

This work is in progress and we plan to enhance our tool by incorporating facilities to support a number of different architectural specifications. Right now, the prototype can handle, uniform and non-uniform unidirectional and bidirectional rings and undirected linear array of processors (chains). We also plan to incorporate limited validation capabilities which we discuss in this abstract briefly.

2 Self-Stabilizing Protocols

The concept of **self-stabilization**, introduced by Dijkstra [Dij74], has been of considerable interest to researchers in the area of fault-tolerant distributed systems. Self-stabilization provides a uniform approach to fault-tolerance [Sch93]. Due to transient faults or arbitrary initialization, a distributed system may enter an undesirable or **illegitimate** global state [Dij74]. In such situations, a self-stabilizing algorithm (protocol) enables the system to recover to a legitimate global state in a finite amount of time. Self-stabilizing algorithms have been developed for a number of problems (see [Sch93, EK89] and the references cited therein).

A Self-Stabilizing protocol design problem is the problem of designing a distributed algorithm such that when the system stabilizes, the resulting global state satisfies a specified requirement. As an example, consider the problem of 2-coloring an even ring. Here an unoriented ring with an even number of nodes is the processor configuration. The requirement is that all the nodes of the ring are colored either 1 or 0 and no two adjacent nodes have the same color. We are required to design a distributed algorithm that will restore the system to a state where the colors assigned to the nodes satisfy the 2-coloring requirement. Moreover, the algorithm must enable the system to reach such a state from *any* initial state.

A distributed algorithm or protocol is **uniform** , if each processor in the distributed system executes the same program. Uniform self-stabilizing algorithms (USSAs) are known for some problems including 6-coloring planar graphs [GK93], finding centers and medians for trees [KPBG94], 2-coloring certain rings and chains [SRR94], orienting odd-length rings [Hoe94], and leader election in rings of prime size [Hua93]. For some problems, it has been shown [Ang80, BP89, IJ93] that *deterministic* uniform self-stabilizing algorithms (DUSSAs) are impossible because of the difficulties encountered in deterministic **symmetry breaking**. For several such problems, researchers have presented randomized algorithms that self-stabilize with high probability; see for example, [Her90, IJ93, SRR94].

It is generally desirable to develop DUSSAs for problems on anonymous networks rather than non-uniform algorithms for ID-based networks. Unfortunately, for anonymous networks, DUSSAs are impossible even for very simple problems (e.g., 2-coloring an anonymous network [SRR94]).

However, as mentioned earlier, designing and then proving correctness of self-stabilizing protocols is a formidable task for even very simple problems. That prompted us to designing a simulation and validation tool for self-stabilizing systems.

A simulation and validation tool for self-stabilizing protocol is very useful in debugging and validating self-stabilizing protocols. There are existing simulation and validation tools for distributed protocols. However, none of them have any special interface for self-stabilizing protocols. Some of the essential features that a simulator for self-stabilizing protocols should have are as follows: arbitrary initialization of the protocol during simulation, facility to describe the protocol at an abstract level with out getting into the implementational details specific to architectural issues, etc.

In this project we use one of the widely available tool (SPIN [Hol90]) for simulation of distributed protocols and build on top of it a simulation tool for self-stabilizing protocols. We design a description language for self-stabilizing protocols which facilitates concise description of the protocols with architectural information. We define the semantics for such a language and also provide translator from this language to the specification language PROMELA which is used for modeling protocols in SPIN. This frontend to SPIN achieves some of our objectives in creating a tool meant for simulation and validation of self-stabilizing protocols. Note that an expert user of SPIN would have done the same by hand encoding the work that is being done by our preprocessor. But that requires knowledge about SPIN and PROMELA. But researchers in the area of self-stabilization mostly used very simple model of a distributed systems. It is shared memory model used in Dijkstra's original work [Dij74]. Also the syntax of describing these protocols have been mostly similar to guarded command language like constructs. The objective of this simplicity in models and language construct is to keep the unessential details minimal so that the understanding of the complex behaviours of such systems is not blurred. We wanted to provide the researchers this facility so that they can specify their self-stabilizing algorithms as simply as they present them in the literature.

It has been common in the literature to express distributed algorithms with a syntax similar to the the Guarded Command language of Dijkstra [Dij76, DF88]. In the description of self-stabilizing distributed algorithms, a similar practice has been followed by many researchers. For example, see [Dij74, GK93, Hua93, Her90, BP89, AG92, SRR94, SRR95, KPBG94, Sch93, AB93, BGM93, EK89, FD94, Hoe94]. Further, many of the self-stabilizing distributed algorithms are designed for specific architectures. For example, consider, self-stabilizing token ring [Dij74], self-stabilizing leader election in odd size rings [Hua93], self-stabilizing median finding in trees [KPBG94], self-stabilizing coloring and other problems for rings, chains and bipartite graphs [SRR94, SRR95]. Notice that the description language used by the research community in presenting these protocols are at a level of abstraction where there is no need to describe the details of the data structuring and other architectural parameters. Mostly, the architecture for which the protocol is designed is mentioned followed by the protocol at an abstract level. As a result, a simulator for such protocols should be able to accept a description or model of the protocol which is at a similar level of abstraction.

More over, if the syntax of the input language to the simulator is similar to the syntax of the language used in describing these algorithms by researchers, and if there is facility for declaring the architecture for which the algorithm is designed, then that facilitates a direct simulation of the protocols described in a very abstract way. This is useful in testing and debugging via simulation and making validation models for these distributed algorithms which are otherwise prone to subtle errors.

3 Short Description of the Project

Here, we describe the design of an input language for our simulator and a prototype implementation of a translator of that input language to a standard process description language (for which simulator and validator are available). The existing widely available tools for simulation of distributed protocols, do not have the facility required by us, to the best of our knowledge.

To accomplish our goal we selected a widely available simulation and validation tool for distributed protocols, called **SPIN**, developed by Gerard J. Holzmann at *AT&T*. The specification language for this tool is called **PROMELA** or PROcess MEta LAnguage [Hol90, Hol93]. Our interest here is to design our simulator on top of this one to simulate or validate architecture dependent protocols with the option of arbitrary initialization.

For example, consider Dijkstra's self-stabilizing token ring protocol [Dij74]. A PROMELA specification of that protocol (for a ring of size 10) is shown in Appendix A. For arbitrary initialization, one has to manually change the initialization in the PROMELA code every time. Moreover, note that, to make sure that the appropriate architecture is assumed by the simulator, detailed design of the global data structure has been made. One can easily see that the local states of each process is simulated as a position in a global array indexed by the process numbers. The variable *state* in each process is now called *state[index]* and when a process refers to a state variable of its left neighbor, it actually refers to *state[(index-1)mod size]* where *size* is the size of the ring. This kind of low level details are required to make sure that unidirectional ring structure is simulated. Also notice that PROMELA needs specification of an *init* process that will start running all the processes. It is clear that one loses the abstraction by making such a low level specification and maintaining data structures specific to the simulator and irrelevant to the correctness of the designed protocol. Hence, if it is possible to design a language that will allow a user to specify protocols at a higher level of abstraction, then it will be easier to use. For example, the protocol in Appendix A can be specified as follows in our language (We called this language **gc**).

```
UNRING(10,P,Q);
PROCESS P{
byte state;
$< state = state(left) >$ -> $<state = state +1 mod 10 >$;
}
PROCESS Q{
$< state != state(left) >$ -> $<state = state(left) >$;
```

```
}
```

Our translator takes this specification and generates low level detailed PROMELA code specific to the SPIN simulator. This relieves the protocol designer from the responsibility of getting involved into unnecessary details specific to SPIN and PROMELA and allows one to concentrate on the correctness at the level of abstraction intended. Moreover, our translator uses random number generator to make arbitrary initialization in the *init* process in each separate compilation. Hence, to simulate distinct arbitrary initializations, one can compile the code several times, each time the resulting PROMELA code will have the effect of a distinct arbitrary initialization.

4 Extensions for Validating Self-Stabilizing Protocols

Most of the research papers on self-stabilizing protocols describe protocols that are parameterized by the number of processors. Hence a correctness proof entails proving the stabilizability irrespective of the number of processors to be used. To automatize such correctness proofs one might use theorem provers but that would require a lot of user interaction. However, from the correctness condition from an implementation point of view, it might be okay to prove the corretness for a specific number of processors. This specific number is exactly the number of processors on which the system is being implemented.

So if f characterizes the legitimate global state, then we want to prove that always eventually f holds and once f holds, it remains stable. In the language of linear time temporal logic this would be $GFf \wedge G(f \rightarrow Gf)$. Thus our frontend will accept the predicate that characterizes legitimacy and then it will generate the LTL formula as above. The frontend complier will insert a **never** claim in the PROMELA file that will be submitted to SPIN.

We are also considering some inductive proof methods similar to [KM89, Kur94] which will help us to prove the correctness for any number of processors.

Another extension that we have planned is to incorporate the adversary assumption in our input language. The user can specify what kind of adversary is assumed (central scheduler, distributed scheduler or other stronger adversaries). The frontend compiler will generate different PROMELA code for each case.

5 Conclusion

The detailed syntax and semantics of our description language will appear in a fuller version of this paper. The syntax is closer to the semantics of the informal algorithmic description languages used for presenting self-stabilizing protocols in the literature. Also, the details of the translation procedure, architectural specifications supported at the moment and future enhancements in that regard will appear in the full version of the paper. In Appendix B we have incorporated a number of examples of self-stabilizing protocols from [Dij74, SRR94, SRR95] being described in our language and the corresponding PROMELA specifications generated by our translator. It should be noted that the result of translating into PROMELA code might not be optimized. For example, instead of using `init` on 50 different processes we could have used a PROMELA feature `active [50] proctype A()` which would have created 50 instances. In the next version of the project we plan to incorporate such optimizations.

Acknowledgement: We wish to thank Mohamed Gouda for interesting discussions and Gerard Holzmann for valuable comments on a previous version of the draft.

References

[AB93] Y. Afek and G. M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7:27–34, November 1993.

[AG92] M.S. Abadir and M.G. Gouda. The stabilizing computer. In *Proceedings of the 1992 international conference on Parallel and Distributed systems.*, 1992.

[Ang80] D. Angluin. Local and Global properties in networks of processes. In *Proceedings of the 12th Annual ACM symposium on theory of computing*, University of California at Los Angeles, 1980. ACM.

[BGM93] J. E. Burns, M. G. Gouda, and R. E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7:35–42, November 1993.

[BP89] J. E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.

[DF88] E. W. Dijkstra and W. H. J. Feijen. *A Method of Programming.* Addison-Wesley Publishing Co., 1988.

[Dij74] E. W. Dijkstra. Self-stabilizing systems inspite of distributed control. *Communications of ACM*, 17(11):643–644, 1974.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall Series in Automatic Computation. Prentice Hall, Inc., 1976.

[EK89] M. Evangelist and S. Katz, editors. *Proceedings of the MCC Workshop on Self-Stabilizing Systems.* MCC, Austin, TX, 1989.

[FD94] M. Flatebo and A. K. Datta. Two-state self-stabilizing algorithms for token rings. *IEEE Transactions on Software Engineering*, June 1994.

[GK93] S. Ghosh and M. H. Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7(1):55–59, November 1993.

[Gou96] M. Gouda. Private communications. May 1996.

[Her90] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[Hoe94] J.-H. Hoepman. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. Technical Report CS-R9423, Computer Science Department, CWI, Amsterdam, April 1994.

[Hol90] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software series, Engelwood Cliffs, NJ, 1990.

[Hol93] G. J. Holzmann. Design and validation of protocols: A tutorial. *Computer Networks and ISDN Systems*, 25(9):981–1017, 1993.

[Hua93] S. T. Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, 15(3):563–573, July 1993.

[IJ93] J. Israeli and M. Jalfon. Self-stabilizing ring orientation. *Information and Computation*, 104(2):175–196, 1993.

[KM89] R. P. Kurshan and K. P. McMillan. A structural induction theorem for processes. In *Proceedings of Conference on Principles of Distributed Computing (PODC)*, pages 239–247. ACM, 1989.

[KPBG94] M.H. Kaarata, S.V. Pemmaraju, S.C. Bruell, and S Ghosh. Self-stabilizing Algorithms for Finding Centers and Medians of Trees. TR- 94-03, University of Iowa, 1994.

[Kur94] R. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton Series in Computer Science. Princeton University Press, 1994.

[Sch93] M. Schneider. Self-stabilization. *Computing Surveys*, 25(1):45–67, March 1993.

[SRR94] S. K. Shukla, D.J. Rosenkrantz, and S.S. Ravi. Developing Self-Stabilizing Coloring Algorithms via Systematic Randomization. In *Proceedings of the 1st International Workshop on Parallel Processing*, pages 668–673, New Delhi, 1994. Tata McGraw-Hill.

[SRR95] S. K. Shukla, D.J. Rosenkrantz, and S.S. Ravi. Observations on self-stabilizing graph algorithms. In *2nd Workshop on Self-stabilizing Systems*, Las Vegas, 1995. University of Nevada at Las Vegas Tech. Report.

A PROMELA Specification of Dijkstra's Token Ring Protocol

```
#define N  10
#define size 10

byte state[size];



proctype A(short index)
{
    do
    :: atomic{ state[index] == state[N-1] ->
                                if
                                 :: (state[index] + 1 == 10) -> state[index] = 0
                                 :: (state[index] + 1 < 10) ->
                                            state[index] = state[index] +1
                                 fi
}
    od
}

proctype B(short index)
{
  do
  :: atomic{ (state[index] != state[index -1]) ->
                                  state[index] = state[index -1]
}
  od
}
init
{
  atomic{ state[0] = 1; state[1] = 2; state[2] = 3; state[4] = 4; state[5] = 5;
state[6] = 3; state[7] = 4; state[8] = 6; state[9] = 9};
atomic{ run A(0); run B(1); run B(2); run B(3); run B(4); run B(5); run B(6);
run B(7); run B(8); run B(9)}
```

```
}
```

B Examples of Translations

B.1 Dijkstra's Token Ring on a 50 processor unidirectional Ring

Here is the **gc** specification of Dijkstra's Token Ring algorithm to be run on a 50 processor ring.

```
RING(50,P,Q);
int state;

PROCESS P{
$<state = state(left) >$ -> $<state := state + 1 >$
}
PROCESS Q{
$<state != state(left) >$ -> $<state := state(left) >$
}
.
```

Here is the PROMELA code generated by our system:

```
#define size 50
int state[50] ;

proctype P(int index){
do
:: atomic{  state[index] == state[(size + index - 1) % size]   ->
                                     state[index] = state[index] + 1   }


od
 }
proctype Q(int index){
do
:: atomic{  state[index] != state[(size + index - 1) % size]   ->
```

```
                        state[index] = state[(size + index - 1) % size]   }

od
 }
init{atomic{state[0] = 2; state[1] = 3; state[2] = 0; state[3] = 1;
 state[4] = 2; state[5] = 3; state[6] = 0; state[7] = 1; state[8] = 2;
 state[9] = 3; state[10] = 0; state[11] = 1; state[12] = 2; state[13] = 3;
 state[14] = 0; state[15] = 1; state[16] = 2; state[17] = 3; state[18] = 0;
 state[19] = 1; state[20] = 2; state[21] = 3; state[22] = 0; state[23] = 1;
 state[24] = 2; state[25] = 3; state[26] = 0; state[27] = 1; state[28] = 2;
state[29] = 3; state[30] = 0; state[31] = 1; state[32] = 2; state[33] = 3;
state[34] = 0; state[35] = 1; state[36] = 2; state[37] = 3; state[38] = 0;
state[39] = 1; state[40] = 2; state[41] = 3; state[42] = 0; state[43] = 1;
state[44] = 2; state[45] = 3; state[46] = 0; state[47] = 1; state[48] = 2;
state[49] = 3;  };

 atomic{run P(0); run Q(1); run Q(2); run Q(3); run Q(4); run Q(5);
 run Q(6); run Q(7); run Q(8); run Q(9); run Q(10); run Q(11); run Q(12);
 run Q(13); run Q(14); run Q(15); run Q(16); run Q(17); run Q(18); run Q(19);
 run Q(20); run Q(21); run Q(22); run Q(23); run Q(24); run Q(25); run Q(26);
 run Q(27); run Q(28); run Q(29); run Q(30); run Q(31); run Q(32); run Q(33);
 run Q(34); run Q(35); run Q(36); run Q(37); run Q(38); run Q(39); run Q(40);
run Q(41); run Q(42); run Q(43); run Q(44); run Q(45); run Q(46); run Q(47);
  run Q(48); run Q(49); }
}
```

We ran this promela file on the SPIN simulator and simulated the algorithm on 50 processor Ring. The novelty is that the user need not know the PROMELA specification language to do the simulation. The **gc** specification is almost the same as the specification a researcher puts in the paper when one develops a distributed algorithm in shared memory model.

B.2 Our Maximal Independent Set Algorithm on a Ring of size 20

Here is the **gc** specification of the system to be simulated.

```
RING(20,P,Q);
bit Ind;
```

```
PROCESS P{
$< (Ind(left) = 0) && (Ind(right) = 0) && (Ind = 0) >$ -> $< Ind := 1 >$
|
$< ((Ind(left) = 1) || (Ind(right) = 1)) && (Ind = 1) >$ -> $<Ind := 0 >$
}

PROCESS Q{
$< (Ind(left) = 0) && (Ind(right) = 0) && (Ind = 0) >$ -> $< Ind := 1 >$
|
$< ((Ind(left) = 1) || (Ind(right) = 1)) && (Ind = 1) >$ -> $<Ind := 0 >$
}
```

Here is the PROMELA Specification generated by our system:

```
#define size 20
bit Ind[20] ;

proctype P(int index){
do
:: atomic{  ( Ind[(size + index - 1) % size] == 0 ) &&
                  ( Ind[(index + 1) % size] == 0 ) && ( Ind[index] == 0 )   ->
                                                            Ind[index] = 1   }
:: atomic{  ( ( Ind[(size + index - 1) % size] == 1 ) ||
                 ( Ind[(index + 1) % size] == 1 ) ) && ( Ind[index] == 1 )  ->
                                                            Ind[index] = 0   }


od
 }
proctype Q(int index){
do
:: atomic{  ( Ind[(size + index - 1) % size] == 0 ) &&
                ( Ind[(index + 1) % size] == 0 ) && ( Ind[index] == 0 )   ->
                                                             Ind[index] = 1   }
```

```
:: atomic{  ( ( Ind[(size + index - 1) % size] == 1 ) ||
              ( Ind[(index + 1) % size] == 1 ) ) && ( Ind[index] == 1 )   ->
                                                           Ind[index] = 0   }

od
 }
init{atomic{Ind[0] = 1; Ind[1] = 1; Ind[2] = 0; Ind[3] = 0; Ind[4] = 1;
 Ind[5] = 1; Ind[6] = 0; Ind[7] = 0; Ind[8] = 1; Ind[9] = 1; Ind[10] = 0;
 Ind[11] = 0; Ind[12] = 1; Ind[13] = 1; Ind[14] = 0; Ind[15] = 0; Ind[16] = 1;
  Ind[17] = 1; Ind[18] = 0; Ind[19] = 0;  };

 atomic{run P(0); run Q(1); run Q(2); run Q(3); run Q(4); run Q(5);
 run Q(6); run Q(7); run Q(8); run Q(9); run Q(10); run Q(11); run Q(12);
 run Q(13); run Q(14); run Q(15); run Q(16); run Q(17); run Q(18); run Q(19);
         }}
```

We ran this also on the SPIN simulator and found the maximal independent set.

B.3 Coloring of Odd chains on a chain of Size 11

Here is the **gc** specification of the coloring of the odd chain of length 11.

```
CHAIN(11, P, Q);
byte X, color;

PROCESS P{
$< X != 0 || color(index) != 0  >$ -> $<X := 0; color := 0 >$
}

PROCESS Q{
$< ((X(left) >= X(right)) && (X != X(right) + 1)) ||
                                          ( color != X %2)  >$ ->
                                $<X := X(right) + 1; color := X %2 >$
|
$< ((X(left) < X(right)) && (X != X(left) + 1)) ||
                    ( color := X %2) >$ ->
                                 $<X := X(left) + 1; color := X %2 >$
```

```
}
.
```

Here is the PROMELA Specification generated by our system:

```
#define size 11
byte X[11] , color[11] ;

proctype P(int index){
do
:: atomic{  X[index] != 0 || color[index] != 0   ->
                                   X[index] = 0 ; color[index] = 0   }

od
 }
proctype Q(int index){
do
:: atomic{  ( ( X[(index -1)] >= X[(index + 1)] ) &&
                     ( X[index] != X[[(index + 1)] + 1 ) ) ||
                                 ( color[index] != X[index] % 2 )   ->
             X[index] = X[(index + 1)] + 1 ; color[index] = X[index] % 2   }
:: atomic{  ( ( X[(index -1)] < X[(index + 1)] ) &&
                      ( X[index] != X[(index -1)] + 1 ) ) ||
                                ( color[index] = X[index] % 2 )   ->
               X[index] = X[(index -1)] + 1 ; color[index] = X[index] % 2   }

od
 }
init{atomic{X[0] = 0; X[1] = 0; X[2] = 4; X[3] = 1; X[4] = 4; X[5] = 4;
 X[6] = 2; X[7] = 0; X[8] = 1; X[9] = 2; X[10] = 3;
  color[0] = 1; color[1] = 1; color[2] = 1; color[3] = 3;
   color[4] = 1; color[5] = 1; color[6] = 4; color[7] = 2;
                 color[8] = 4; color[9] = 2; color[10] = 0;  };
```

```
atomic{run P(0); run P(10);run Q(1); run Q(2); run Q(3); run Q(4);
       run Q(5); run Q(6); run Q(7); run Q(8); run Q(9); }}
```