Model and Verify CPS using "Büchi Automata"

Enkeledi Mema[1]

Contents

Abstract: In this paper is described a general notation of the Büchi Automata(BA). A brief definition and comparison between BA and Normalised Büchi Automata (NBA) is provided. Efficient model checking algorithms for linear time temporal logic were developed by exploiting the close relationship of the temporal logic and automaton on infinite words. Furthermore, formal verification of CPS were done. A transformation from linear temporal logic (LTL) to Normalised Büchi Automata is presented by using different tools. Spin,Spot LTL2TGBA and LTL3BA are used as tools to convert on the process.Verification and validation were done for the CPS.

[1] enkeledi.mema@stud.hshl.de

1 Motivation

Reactive systems are present in nowadays. This kind of system interacts with their environment via inputs and outputs.To perform model checking is crucial and is becoming a more difficult task. Therefore, model checking is very important. The needs for these types of theoretical machines to verify different models are excessive[VB]. Lately such kind of systems as CPS are widely used in areas like smart cities, medicine industry, avionic industry, traffic control and safety, process control, etc. In CPS we have two sides. In one side are cooperating in computational systems including software and hardware and in the other side is a physical system. Combining and developing this kind of smart systems is a difficult task. Moreover, to verify and validate them are becoming more difficult. For example, in a safety critical scenario. This type of CPS relies less on human supervision and is becoming more dependable. The goal is to make such kind of systems safe and secure. The CPS might have several inputs and processes, this end up in a extremely complex interaction [Gr].

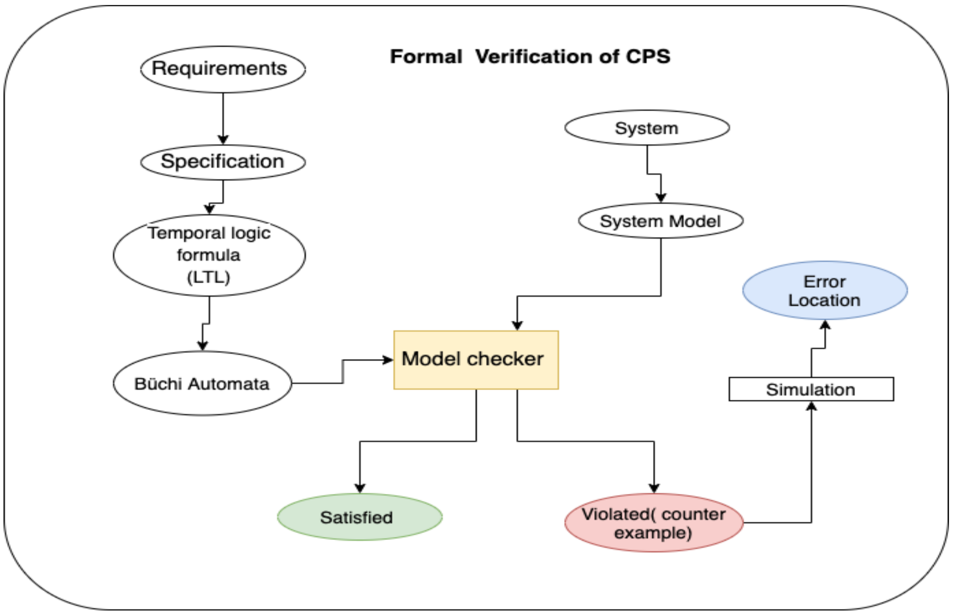


Fig. 1: System Architecture of CPS For Model Checking.

Therefore, the chances for failures are extraordinary, and they can happen at any time. Taking into consideration these properties, verification and validations plays a crucial role. Decades ago, this formal verification was performed more on the hardware.Lately, they are used both in hardware and software. This type of verification is developed by creating a mathematical computer-based system. This mathematical system verifies rigorously if a property is satisfied[Go]. In this way we make sure the system is performing the exact thing and satisfies the requirements. The formal verification are relatively beneficial tools to detect critical design errors, that are hard to detect by naked human eye and manual proofs.For example, on January 1990 in USA was a telephone outage and costed around one hundred million dollars. The outage was caused by wrong interpretation of a break statement in a loop c language. This leaded to a bug that by using paper and pencil method to catch it was impossible. After formal verification it was realised the incorrect interpretation. If we place the focus in safety critical systems, we understand that the formal verification carry out a significant role[Me02].

2 Foundation

2.1 Introduction to Büchi Automata

The Büchi automata is capable to accept infinite inputs[MR]. BA has a set of initial states and could generate multiple transitions from a state. A specific input may lead in different outputs therefore might be present different paths. The main purpose of BA is to generate deterministic and non-deterministic finite automaton from infinite inputs.BA are used widely as an automaton theoretic version of a LTL[Fi]. This means that every LTL formula has a Büchi automaton that accepts its language, but this is true in one direction and not vice versa. It is widely used because of omega regular languages acceptance. The key goal of using BA it that it can translate model checking of finite systems into different operations. This ability of BA is powered by closure operations that BA poses[EL]. The main operations for BA application are three.

- Determinization
- Emptiness checking
- Minimization

2.1.1 Büchi Automaton Definition

Definition 2.1 (Büchi Automata [Ch].) *Let $M(Q,\Sigma,\delta,q_0,F)$ a tuple be a deterministic Büchi automaton*

- *Q is a set of states;*
- *Σ is a finite set called the alphabet of A;*
- *$\delta\ Q\times\Sigma\rightarrow Q$is a transition function of A ;*
- *q_0 is initial state of A ;*
- *F is the acceptance condition ;*

2.1.2 Büchi Automaton and Generalized Büchi Automata Example

Generalized Büchi automaton represent a variant of BA. It differs from accepting condition. We make certain that for the sets of states exist a set. GBA is the same as BA regarding the expressive power. When formal verification are performed for a finite state model checking, there is needs to find a Büchi automaton (BA) comparable to a provided Linear Temporal Logic (LTL) formula[Ch]. For example, an significant case might be that LTL formula and the BA recognize the same omega language. There are algorithms that translate an LTL formula to a BA[AAF]. To perform this transformation, we need to go through two steps. The first step is to create a GBA from a LTL formula.Therefore using GBA is a superior approach for model checking. The second step is to translate this GBA into a BA, which is practically an easy construction.The algorithms for transforming LTL to GBA differ in their construction strategies.To realize this transformation and constructions, various tools and algorithms are available[MR].

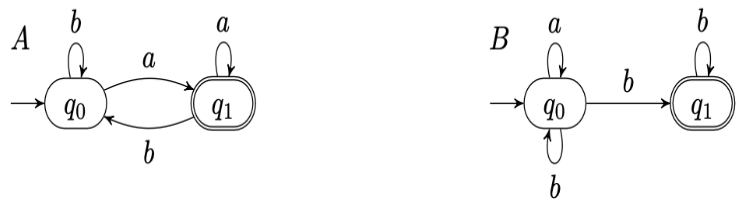


Fig. 2: A is a Deterministic Büchi Automaton and B is Generalized Büchi Automaton[Li].

In the picture A we have a deterministic Büchi automaton.In this case we have $\Sigma = \{a, b\}$ which is the alphabet.The set of the transition $\delta = \{(q0, a, q1), (q0, b, q0), (q1, a, q1), (q1, b, q0)\}$. The acceptance condition $F = \{q1\}$ and the initial state $q_0 = \{q0\}$. In the picture B is an NBA that accepts ω-language and has finitely many a's, which are the complement of the automaton in picture A. In the other side automaton in picture A has infinitely many a's[Li].

2.2 Use Case of CPS Verification and Validation

Previously is mentioned that CPS can be used widely in smart cities. One scenario might be managing the traffic light.For example in case of emergency the police or ambulance can change the traffic lights to green and they can save time. In this condition time is very important and every second is taken into consideration.This kind of system can be interpreted as a CPS.In every cross section there are traffic lights. We can assign an ID to each cross section and we can control and supervise them remotely.We can combine information from CCTV ,Sensors in cross section and traffic lights .We can have them in central office. Now the traffic lights is modeled as virtual object in the network.In one hand we have the physical system (Cross-Section) and in the other hand we have the virtual system.Interpreting this system as CPS solves many things and saves lives.This kind of CPS are difficult to be build because we have to deal with safety critical systems.Therefore verification and validation are important.

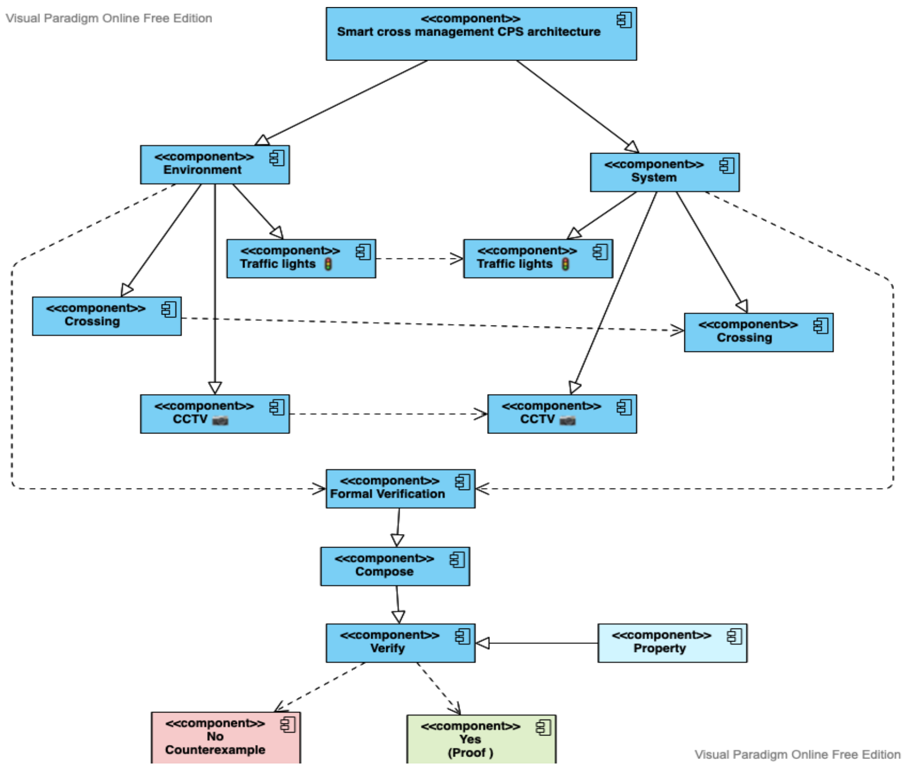

Visual Paradigm Online Free Edition
<<component>> Smart cross management CPS architecture
<<component>> Environment
<<component>> System
<<component>> Traffic lights
<<component>> Traffic lights
<<component>> Crossing
<<component>> Crossing
<<component>> CCTV
<<component>> CCTV
<<component>> Formal Verification
<<component>> Compose
<<component>> Verify
<<component>> Property
<<component>> No Counterexample
<<component>> Yes (Proof)
Visual Paradigm Online Free Edition

Fig. 3: UML Component Diagram for CPS Verification And Validation.

In the picture above is represented the system architecture of this CPS.In one side is the real environment with its own components and in the other side we have the cyber-system.To make sure that the CPS is doing the right thing and is fulfilling the requirements we need to do some verification.In this way we have another component in our system called formal verification.Usually we verify if a property is achieved or not.In case it is achieved we get a positive feedback. In case it is not fulfilled we get a negative response and counterexample is generated[He].

2.3 CPS Use Case Implementation

To implement this kind of verification we need to go through some steps and conversion.First of all we need to specify one property that we are interested to perform verification.The next step is to build an automaton for this property.In our case we will use Büchi Automaton.After we build our automaton, we can convert it to LTL formula.Having the property in LTL is crucial and useful because now we have a computer mathematical expression for our property. Also it is easier to perform model checking using a software.After the conversion is done now we can check our model.In this scenario the Spin model Checker is used.In the meantime we have to take care for the model and software that we need to verify.It is a better idea to combine them an bring them close to spin syntax.We can use C as a low-level language.Also other languages are possible [Al]. As we notice in the picture below we have

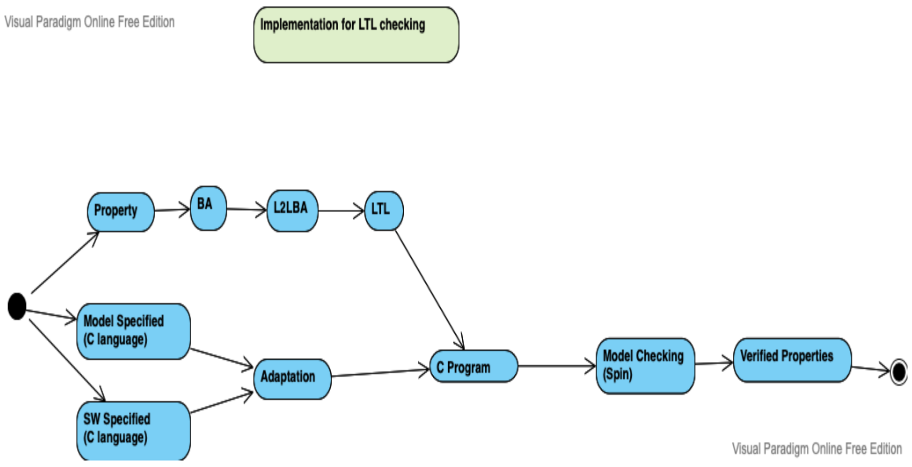

Visual Paradigm Online Free Edition
Implementation for LTL checking
Property
BA
L2LBA
LTL
Model Specified (C language)
SW Specified (C language)
Adaptation
C Program
Model Checking (Spin)
Verified Properties
Visual Paradigm Online Free Edition

Fig. 4: UML Activity Diagram for Implementation.

an UML Activity Diagram to describe the steps and activities that we should follow to implement the verification and model checking for such kind of CPS system.

2.4 Example For The Use Case

As example it is selected a scheduling behavior of the cross section management.The cross section is represented by a grid and this grid is split in small cells.We check if a cell is free or not.Also we check for a sequence of cells.In this case we see that we might have infinite inputs and outputs, in this point verification becomes difficult.Below is the mathematical expression to verify this property. Scheduling implies cell is free and not busy this is the property that we have to model and verify.We have the property in a mathematical representation and it is possible to generate the BA an then LTL or vice versa.In this example is generated the NBA and from the NBA is generated the LTL. The

```
LTL = G ("Schedule" IMP (F "Cell_Free" AND NOT "Cell_Busy")),
ALPHABET = [Cell_Free, Schedule, Cell_Busy]
```

LTL is useful to implement model checking later in Spin.As we see in the picture below the property is modeled in NBA. In this example the alphabet is Cell-Free,Cell-Busy and Schedule.The accepted condition is q0.The initial state we have q2. We can identify q2 as starting point because it has a start arrow.As well we have a set of accepted transition.A scenario of Sequence of transition could be :(q2,Cell-Busy, q0,Cell-Free, ,q0,Schedule, q1),(q2,Cell-Free,q0,Schedule,q1 ,Cell-Busy, q1,Cell-Free ,q0),...etc. In the picture above

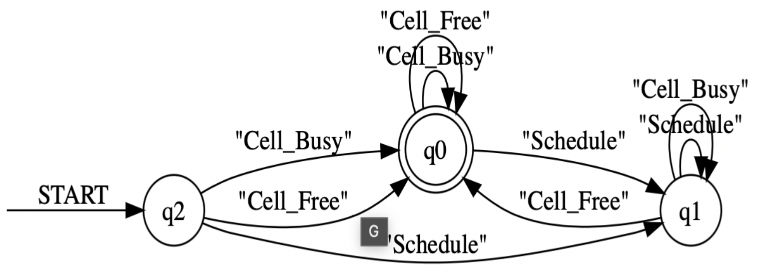
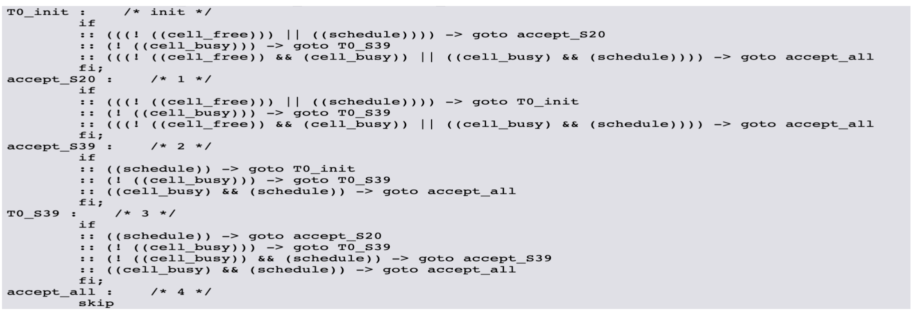


Fig. 5: BA For Formal Verification of The Property.

we can see that after the LTL construction and BA implementation is time to generate a Spin expression that will help us develop examples for verification.

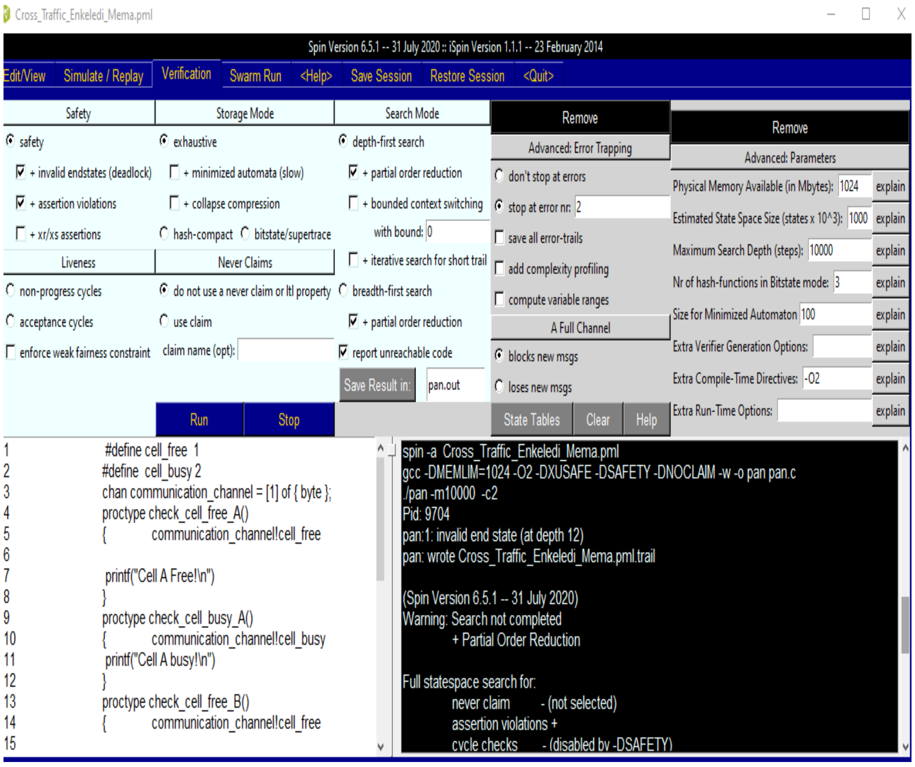


```
T0_init :     /* init */
        if
        :: (((! ((cell_free))) || ((schedule)))) -> goto accept_S20
        :: (! ((cell_busy))) -> goto T0_S39
        :: (((! ((cell_free)) && (cell_busy)) || ((cell_busy) && (schedule)))) -> goto accept_all
        fi;
accept_S20 :     /* 1 */
        if
        :: (((! ((cell_free))) || ((schedule)))) -> goto T0_init
        :: (! ((cell_busy))) -> goto T0_S39
        :: (((! ((cell_free)) && (cell_busy)) || ((cell_busy) && (schedule)))) -> goto accept_all
        fi;
accept_S39 :     /* 2 */
        if
        :: ((schedule)) -> goto T0_init
        :: (! ((cell_busy))) -> goto T0_S39
        :: ((cell_busy) && (schedule)) -> goto accept_all
        fi;
T0_S39 :     /* 3 */
        if
        :: ((schedule)) -> goto accept_S20
        :: (! ((cell_busy))) -> goto T0_S39
        :: (! ((cell_busy)) && (schedule)) -> goto accept_S39
        :: ((cell_busy) && (schedule)) -> goto accept_all
        fi;
accept_all :     /* 4 */
        skip
```

Fig. 6: BA Converted to Spin.

2.4.1 Results Of The Example

After we generate the Automaton we convert it to spin syntax .Now is possible to verify our property. In the picture below is introduced scheduling of two cells A and B. We try to check the state of cell by using some Communication channels.An atomic execution of prototypes is done to make sure we execute in the same time all processes.After the verification we can check if the property is satisfied or not.This kind of verification are difficult or impossible to be performed by pen and paper.Also might be bugs on the code or bad interpretation of statements which can not be detected with naked eye. This errors might cause casualties. Further information can be found in the appendix for verification and simulation in Spin.



Safety
[✓ safety]
[✓] + invalid endstates (deadlock)
[✓] + assertion violations
[ ] + xr/xs assertions

Storage Mode
[✓ exhaustive]
[ ] + minimized automata (slow)
[ ] + collapse compression
hash-compact bitstate/supertrace

Search Mode
[✓ depth-first search]
[✓] + partial order reduction
[ ] + bounded context switching
with bound: 0
[ ] + iterative search for short trail
breadth-first search
[✓] + partial order reduction
[✓] report unreachable code
Save Result in: pan.out

Liveness
non-progress cycles
acceptance cycles
[ ] enforce weak fairness constraint

Never Claims
[✓ do not use a never claim or ltl property]
use claim
claim name (opt):

Advanced: Error Trapping
don't stop at errors
[✓ stop at error nr: 2]
[ ] save all error-trails
[ ] add complexity profiling
[ ] compute variable ranges

A Full Channel
[✓ blocks new msgs]
loses new msgs

Advanced: Parameters
Physical Memory Available (in Mbytes): 1024
Estimated State Space Size (states x 10^3): 1000
Maximum Search Depth (steps): 10000
Nr of hash-functions in Bitstate mode: 3
Size for Minimized Automaton 100
Extra Verifier Generation Options:
Extra Compile-Time Directives: -O2
Extra Run-Time Options:

```
 #define cell_free  1
#define  cell_busy 2
chan communication_channel = [1] of { byte };
proctype check_cell_free_A()
{           communication_channel!cell_free

 printf("Cell A Free!\n")
}
proctype check_cell_busy_A()
{           communication_channel!cell_busy
 printf("Cell A busy!\n")
}
proctype check_cell_free_B()
{           communication_channel!cell_free
```

```
spin -a  Cross_Traffic_Enkeledi_Mema.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000  -c2
Pid: 9704
pan:1: invalid end state (at depth 12)
pan: wrote Cross_Traffic_Enkeledi_Mema.pml.trail

(Spin Version 6.5.1 -- 31 July 2020)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim         - (not selected)
        assertion violations +
        cycle checks        - (disabled by -DSAFETY)
```

Fig. 7: Spin Verification

3 Conclusion

In this paper is demonstrated a scenario of a CPS verification and validation using omega expression languages such as BA and GBA.As an use case a smart city application is selected. The application consists on implementing the smart traffic management as CPS.A detailed description of the CPS is presented.Based on this system formal verification of specific properties is performed.During the process several tools were involved.Tools were mainly used to generate the BA and GBA and to verify this automaton.To check the model Spin was used as a tool.Also different tools like L2LBA are used between conversion.Also computer mathematical expression such as LTL were involved in the process.This CPS has huge potential to change the traffic behaviour in a smart city. Running this king of verification lead us to more secure,safe and reliable CPS.Further improvements can be done in direction of optimization and error reduction.

4 Declaration of Originality

I, Enkeledi Mema herewith declare that I have composed the present paper and work by myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance. I agree that my work may be checked by a plagiarism checker.

Bibliography

[AAF] Angluin, Dana; Antonopoulos, Timos; Fisman, Dana: Strongly Unambiguous Büchi Automata Are Polynomially Predictable With Membership Queries. p. 17 pages. Artwork Size: 17 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany Version Number: 1.0.

[Al] Alur, Rajeev: Principles of cyber-physical systems. The MIT Press.

[Ch] Chechik, Marsha: Automata-Theoretic LTL Model-Checking. p. 12.

[EL] Eid, Abdulla; Lavalle, Steven M: Finite Omega-Automata and Bu¨chi Automata. p. 17.

[Fi] Fisher, Corey S: A More Robust Corpus of Büchi Automata. p. 47.

[Go] Goos, Gerhard; Hartmanis, Juris; van Leeuwen, Jan; Hutchison, David; Kanade, Takeo; Kittler, Josef; Kleinberg, Jon M; Mattern, Friedemann; Mitchell, John C; Naor, Moni; Nierstrasz, Oscar; Rangan, C Pandu; Steffen, Bernhard: Lecture Notes in Computer Science. p. 646.

[Gr] Greer, Christopher; Burns, Martin; Wollman, David; Griffor, Edward: , Cyber-physical systems and internet of things.

[He] Hempfling, Christina: Julius-Maximilians-Universit¨at Wu¨rzburg Faculty of Mathematics and Computer Science Course of Studies: Computer Science. p. 120.

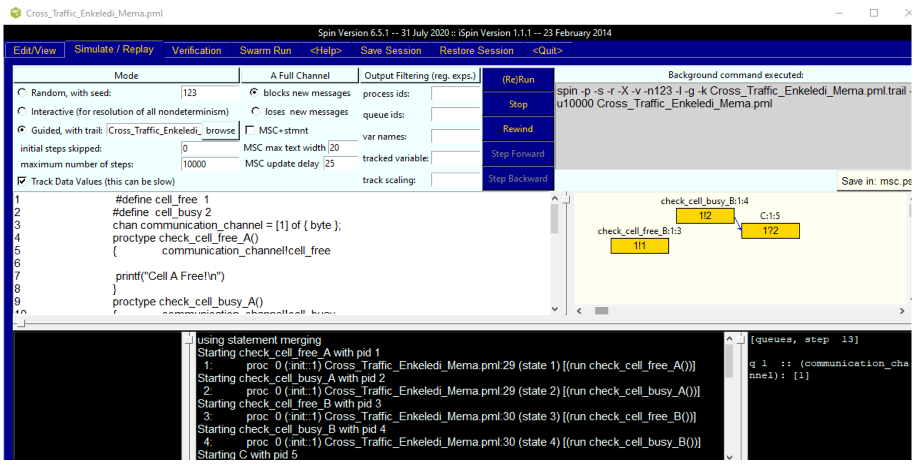
[Li] Li, Yong; Turrini, Andrea; Chen, Yu-Fang; Zhang, Lijun: Learning Büchi Automata and Its Applications. 11430:38–98. Series Title: Lecture Notes in Computer Science.

[Me02] Merz, Stephan: Model Checking Techniqes for the Analysis of Reactive Systems. Synthese, 133(1/2):173–201, 2002.

[MR] Mukund, Madhavan; Road, G N Chetty: Linear-Time Temporal Logic and Bu¨chi Automata. p. 22.

[VB] Visser, Willem; Barringer, Howard: Practical CTL * model checking: Should SPIN be extended? 2(4):350–365.

5 Appendix



Mode

- Random, with seed: 123
- Interactive (for resolution of all nondeterminism)
- [✓ Guided, with trail: Cross_Traffic_Enkeledi_ browse]

initial steps skipped: 0

maximum number of steps: 10000

[✓ Track Data Values (this can be slow)]

A Full Channel

- [✓ blocks new messages]
- loses new messages

MSC+stmnt

MSC max text width 20

MSC update delay 25

Output Filtering (reg. exps.)

process ids:

queue ids:

var names:

tracked variable:

track scaling:

Background command executed:

```
spin -p -s -r -X -v -n123 -l -g -k Cross_Traffic_Enkeledi_Mema.pml.trail -
u10000 Cross_Traffic_Enkeledi_Mema.pml
```

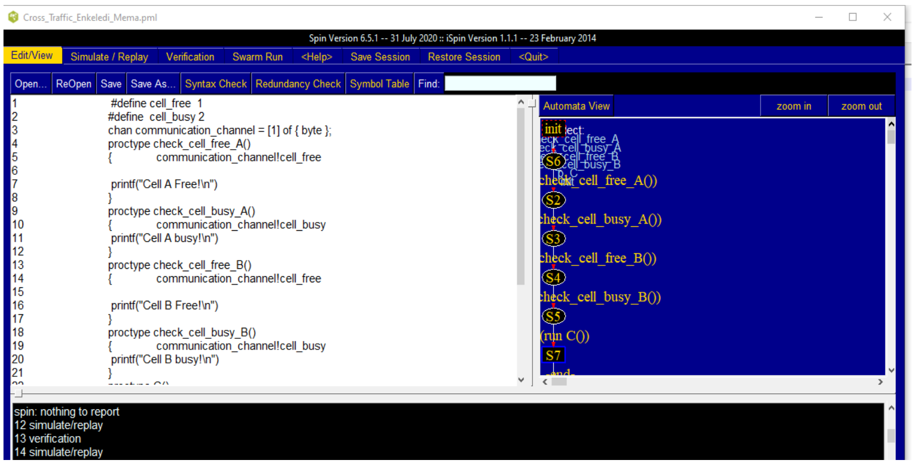
Save in: msc.ps

```
#define cell_free  1
#define  cell_busy 2
chan communication_channel = [1] of { byte };
proctype check_cell_free_A()
{           communication_channel!cell_free

 printf("Cell A Free!\n")
}
proctype check_cell_busy_A()
```

```
using statement merging
Starting check_cell_free_A with pid 1
  1:        proc  0 (:init::1) Cross_Traffic_Enkeledi_Mema.pml:29 (state 1) [(run check_cell_free_A())]
Starting check_cell_busy_A with pid 2
  2:        proc  0 (:init::1) Cross_Traffic_Enkeledi_Mema.pml:29 (state 2) [(run check_cell_busy_A())]
Starting check_cell_free_B with pid 3
  3:        proc  0 (:init::1) Cross_Traffic_Enkeledi_Mema.pml:30 (state 3) [(run check_cell_free_B())]
Starting check_cell_busy_B with pid 4
  4:        proc  0 (:init::1) Cross_Traffic_Enkeledi_Mema.pml:30 (state 4) [(run check_cell_busy_B())]
Starting C with pid 5
```

```
[queues, step  13]

q 1  :: (communication_cha
nnel): [1]
```

Fig. 8: Spin Simulation



```
#define cell_free  1
#define  cell_busy 2
chan communication_channel = [1] of { byte };
proctype check_cell_free_A()
{           communication_channel!cell_free

 printf("Cell A Free!\n")
}
proctype check_cell_busy_A()
{           communication_channel!cell_busy
 printf("Cell A busy!\n")
}
proctype check_cell_free_B()
{           communication_channel!cell_free

 printf("Cell B Free!\n")
}
proctype check_cell_busy_B()
{           communication_channel!cell_busy
 printf("Cell B busy!\n")
}
```

```
spin: nothing to report
12 simulate/replay
13 verification
14 simulate/replay
```

Fig. 9: Spin Automata View