Formalizing Time Petri Nets with Metric Temporal Logic using Promela

Jutamard Kawises
Department of Computer Engineering
Faculty of Engineering, Chulalongkorn University
Bangkok, Thailand
jutamard.ka@student.chula.ac.th

Wiwat Vatanawood
Department of Computer Engineering
Faculty of Engineering, Chulalongkorn University
Bangkok, Thailand
wiwat@chula.ac.th

***Abstract*—Formal verifications for the real-time systems are crucial and inevitable. The formal model and its desirable properties may be checked using model checker. Time Petri net is one of the practical representation for the formal model and its target properties may be efficiently written in metric temporal logic providing the time duration checking. However, the model checking of the time Petri nets with metric temporal logic is still merely supported. In this paper, we propose an alternative to transform the given time Petri nets along with the metric temporal logic into Promela code, which would be practically verified using SPIN tool. We also modified the previous work on translating basic metric temporal logic (MTL) formulas into linear temporal logic (LTL) written in Promela as to cope with the next and until temporal operators. The resulting Promela codes are correctly generated and verifiable using SPIN tool.**

Keywords— Time Petri nets, Metric temporal logic (MTL), Linear temporal logic (LTL), Promela, SPIN Tool

I. INTRODUCTION

In real-time systems, the time restrictions on the timings of events or actions are crucial. Both hard and soft constraints should be satisfied for the system correctness and performance. It would be a practical mean to perform the formal verification of these specifications of the real-time constraints. The use of temporal logics or some formal languages with temporal constructs is essential. One of the formal representation choices of the real-time systems is using the time Petri nets. Moreover, in order to define time constraints in more effective way, the temporal logic would be considered especially the metric temporal logic (MTL). The metric temporal logic is more suitable for the hard and soft constraints with the capability of defining the time intervals [i, j] where i is the starting point and j is the endpoint of the time intervals. However, given a time Petri net with the metric temporal formula, there exist merely practical model checking tools supporting the time constraints. For example, formal modeling with Promela in SPIN tool [1] does not support the model checking for metric temporal formula.

In this paper, we intend to provide an alternative to transform the time Petri nets with the metric temporal formula into Promela codes by extending a previous work [2] on translating basic metric temporal logic formulas into linear temporal logic (LTL) written in Promela.

We propose a transforming of the time Petri nets into Promela code along with the transforming of the MTL formula into some specific Promela code to emulate the checking of the time intervals in MTL formulas.

This paper is organized as follows. Section II presents research background. Section III describes the transforming scheme. Section IV presents our demonstration of case study, and our conclusion of this paper is described in section V.

II. BACKGROUND AND REALTED WORK

A. Metric Temporal Logic[3,4]

Metric temporal logic (MTL) is a particular logic system that is popular for a real-time system and it extends LTL with time intervals. MTL has two main semantics: point-based on timed words and continuous on time flows. It can refer to discrete time in timestamp format.

Given P is a set of atomic propositions, MTL defines over P and truth value under until operator (depicted as U) of event (depicted as φ) as follows:

$$\varphi ::= \rho | \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \ \ U_I \varphi \ , \qquad (1)$$

When $\rho \in P$ and $I \subseteq (0, \infty)$ is time of the end point.

A sample of a MTL formula is shown as follows:

$$\Box(\text{alarm} \Rightarrow (\Diamond_{(0,10)}\text{allClear} \vee \Diamond_{\{10\}}\text{shutdown})) \qquad (2)$$

Which means that every time the alarm event occurs then the shutdown event must occur at t_{10} ($\Diamond_{\{10\}}$) except the allClear event occurs during t_0 - t_{10} at least once.

There were researches on mapping between LTL and MTL. For example, [4] found that there are two fundamental decision problems, the satisfiability problem and the model-checking problem found in MTL semantic model, concerning punctuality, boundedness, flatness and safetyness.

B. Linear Temporal Logic[5,6,7]

Linear temporal logic (LTL) is kind of logic system being capable of modeling time. It encodes a past or future of event occurring. However, LTL is typically used to model the future of event. Also, there are basic temporal connectives or operators for the future temporal logic.

1) Always

The always operator (depicted as □) indicates that the event always occurs henceforth until the end of system times.

2) Eventually

The eventually operator (depicted as ◊) indicates that the event occurs at least once from the present to the future time but do not know whenever.

3) Next

The next operator (depicted as ○) indicates that the event occurs immediately next after the present time.

4) Strong until/ until

The strong until operator (depicted as U) concerns two related events and indicates that the first event must occur from the present time and still occurs until the second event would occur once in the future time.

5) Weak until/ unless

The weak until operator (depicted as W) concerns two related events and indicates that the first event must occur from the present time and still occurs until the second event would occur once in the future time. However, the second event may not occur but the first event remains occurring and waiting for the second event.

$\Diamond_{(t1,t2)}p$

```
#define MTL_Fpp(effect) <>(effect.Reaction&&(effect.EventCounter>
   effect.pTime1)&&(effect.EventCounter<effect.pTime2))
```

$\Diamond_{[t1,t2]}p$

```
#define MTL_Fss(effect) <>(effect.Reaction&&(effect.EventCounter>=
   effect.pTime1)&&(effect.EventCounter<=effect.pTime2))
```

$\Box_{(t1,t2)}p$

```
#define MTL_Gpp(effect) [](!((effect.EventCounter>effect.pTime1)&&
    (effect.EventCounter<effect.pTime2))||((effect.Reaction
   &&(effect.EventCounter>effect.pTime1))&&(effect.EventCounter
   <effect.pTime2)))
```

$\Box_{[t1,t2]}p$

```
#define MTL_Gss(effect) [](!((effect.EventCounter>=effect.pTime1)&&
    (effect.EventCounter<=effect.pTime2))||((effect.Reaction
   &&(effect.EventCounter>=effect.pTime1))&&(effect.EventCounter
   <=effect.pTime2)))
```

Fig. 1. The LTL in 4 conditions.[2].

C. Petri nets

A Petri net [8,9] is a bipartite graph used to model a system process. The Petri net consists of four types of symbols which are place, transition, directed arc and token. A Petri net is defined as a 5-tuple, PN = (P, T, F, W, m_0) where

P is a finite set of places

T is a finite set of transition, $P \cap T = \varnothing$

$F \subseteq (P \times T) \cup (P \times T)$ is the function of flow relation

W is the weight function of flow relation

m_0 is the initial marking of the Petri net

A sample of Petri net is shown in Fig. 2. The system process model would be designed using some particular constructs which represent the patterns of primitive process flows such as sequential, conflict or choice, concurrent, synchronization, mutual exclusive, and priority. The mostly used Petri net's constructs are shown in Fig. 3.

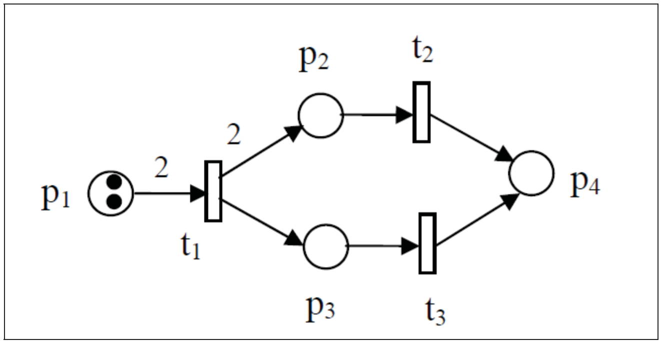


Fig. 2. A Sample of Petri net.

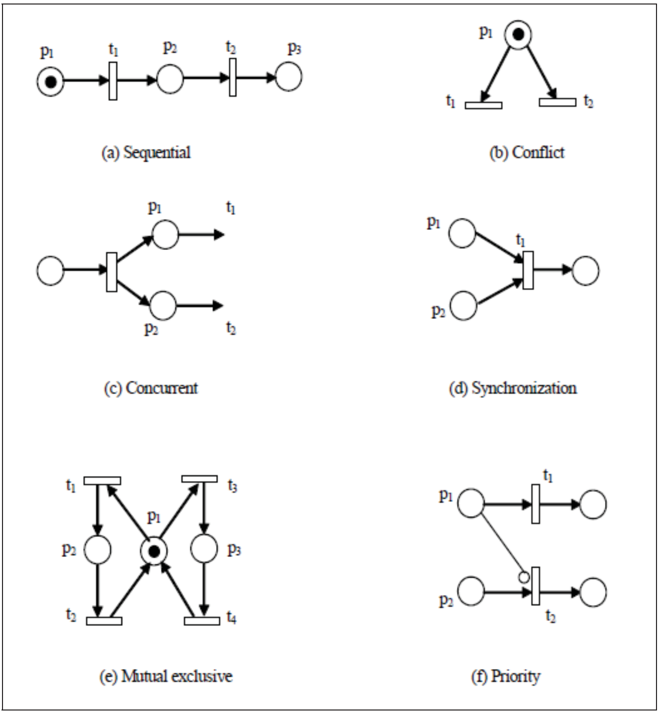


Fig. 3. The Petri net's constructs.[8].

The Petri net's constructs [8] are described as follows.

1) Sequential

The sequential execution indicates that the transition t_1 would be firing and the token would be consumed from p_1 and produced into p_2. Then, the following transition t_2 would be firing subsequently. This construct is used when a task need to performs after one.

2) Conflict

The conflict execution is another not so obvious construct, indicates that the either one transition t_1 or t_2 would be firing non-deterministically.

3) Concurrency

The concurrency execution indicates that the transition would be firing to the more than one post places which allows the concurrency of the subsequent paths from these post places. Many complex systems inevitably contain this concurrent construct as well.

4) Synchronization

The synchronization execution indicates that the more than one pre places are expected to be consumed simultaneously. It means that all pre places would contain the tokens.

5) Mutually Exclusive

The mutual exclusive execution indicates that the either transition t_1 or t_3 would be guarantee not to be firing simultaneously. If transition t_1 is firing then transition t_3 would be disabled.

6) Priorities

The priority execution indicates that there is one transition t_1 would be firing before the others. By introducing an inhibitor arc to disable t_2 if there is a token living in p_1, the transition t_1 would be firing.

D. Time petri nets

A time Petri net is the extension of Petri nets to explain dynamic system, linear sequence of states and transitions with their association of lowerbound and upperbound time intervals. The time interval constraints are introduced to effectively capture the real-time constraints [8, 10]. According to the Petri net PN mentioned earlier, a time Petri net is defined as TPN=(PN, SI) where SI is the static time interval function. A sample of time Petri net is shown in Fig. 4.

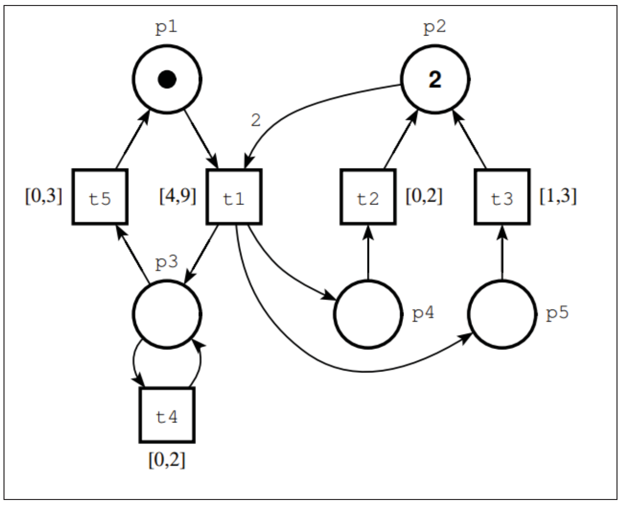


Fig. 4. A sample of time Petri net [11].

III. Our Transforming Scheme

In this section, we propose our transforming scheme to formalize the time Petri net with MTL formula into Promela codes. Firstly, a case study of time Petri net is given. The time Petri net would be transform into the corresponding Promela codes. Secondly, we refer to [2] in order to get how the global clock and local clock implemented and extend [2] to cover the next and strong until operators. Then, the final Promela codes would be consolidated and verified using SPIN tool.

A. The case study of time Petri net

Given a system process represented using a time Petri net shown in Fig. 5. The time Petri net model contains four places and three transitions attached with the local clocks, called sub clocks. Each transition would be time interval stamped as [i,j] where i is the start time and j is the end time. There is a global clock of the system, called main clock.

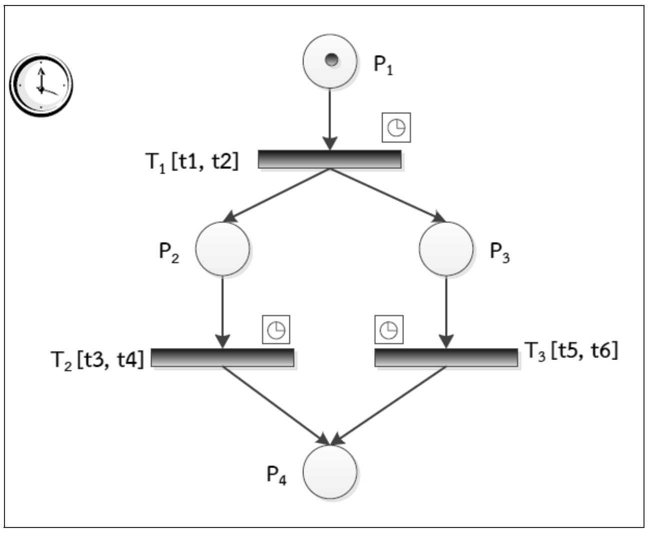
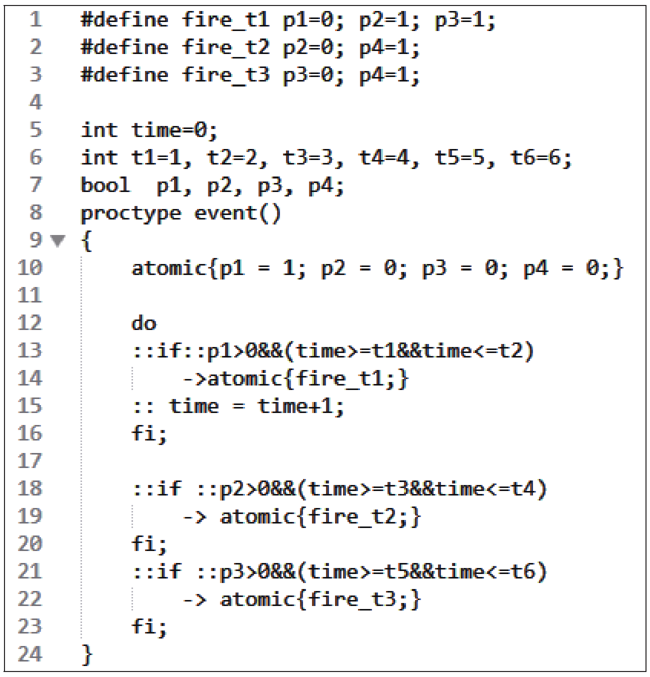


Fig. 5. Model of system process.



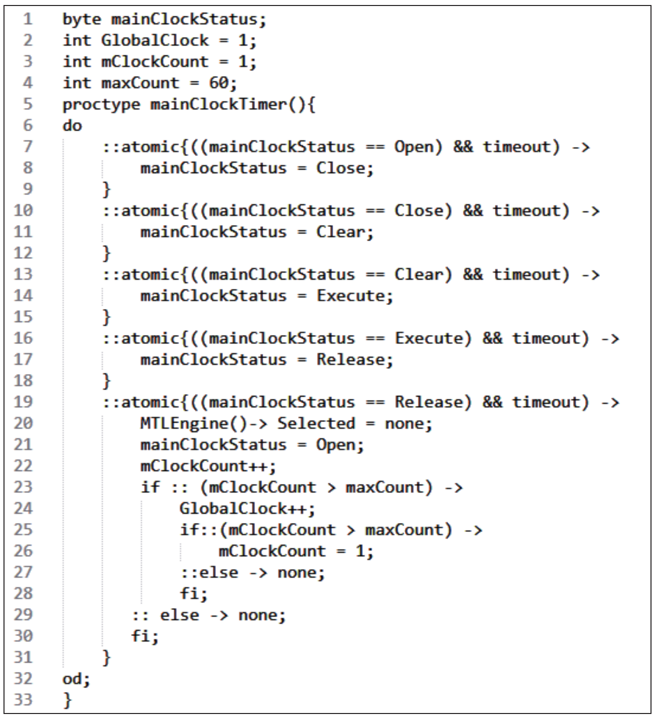
```
#define fire_t1 p1=0; p2=1; p3=1;
#define fire_t2 p2=0; p4=1;
#define fire_t3 p3=0; p4=1;

int time=0;
int t1=1, t2=2, t3=3, t4=4, t5=5, t6=6;
bool  p1, p2, p3, p4;
proctype event()
{
    atomic{p1 = 1; p2 = 0; p3 = 0; p4 = 0;}

    do
    ::if::p1>0&&(time>=t1&&time<=t2)
        ->atomic{fire_t1;}
    :: time = time+1;
    fi;

    ::if ::p2>0&&(time>=t3&&time<=t4)
        -> atomic{fire_t2;}
    fi;
    ::if ::p3>0&&(time>=t5&&time<=t6)
        -> atomic{fire_t3;}
    fi;
}
```

Fig. 6. The Promela of the time Petri net.

The time Petri net from Fig. 5 would be transform into Promela shown in Fig. 6. It defines initial marking for p1 = 1, p2 = 0, p3 = 0 and p4 = 0. After the firing of T1 defined by fire_t1, the next marking would be p1 = 0, p2 = 1, p3 = 1 and p4=0.

B. *The clock timer*

We refer to [2] for the implementation of the clock timer. The main clock is defined in Promela in [2] and we import this protype into our Promela. The implementation of the main clock is shown in Fig. 7. The clock consists of five states which are open, close, clear, execute and release. The clock would be counting from 1 to 60 and reset when counting to maxCount variable. These counting variables would be referred to when the time intervals in MTL are implemented.



```
byte mainClockStatus;
int GlobalClock = 1;
int mClockCount = 1;
int maxCount = 60;
proctype mainClockTimer(){
do
    ::atomic{((mainClockStatus == Open) && timeout) ->
        mainClockStatus = Close;
    }
    ::atomic{((mainClockStatus == Close) && timeout) ->
        mainClockStatus = Clear;
    }
    ::atomic{((mainClockStatus == Clear) && timeout) ->
        mainClockStatus = Execute;
    }
    ::atomic{((mainClockStatus == Execute) && timeout) ->
        mainClockStatus = Release;
    }
    ::atomic{((mainClockStatus == Release) && timeout) ->
        MTLEngine()-> Selected = none;
        mainClockStatus = Open;
        mClockCount++;
        if :: (mClockCount > maxCount) ->
            GlobalClock++;
            if::(mClockCount > maxCount) ->
                mClockCount = 1;
            ::else -> none;
            fi;
       :: else -> none;
       fi;
    }
od;
}
```

Fig. 7. The implementation of the main clock [2].

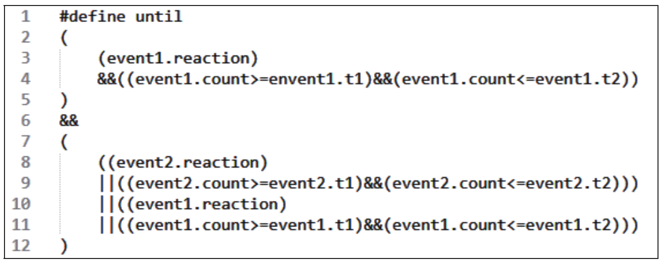
C. *Transforming the MTL into LTL*

In this paper, we extend [2] to cope with the next and strong until temporal operators.

1) *Strong Until*

Given a generic formula of the strong until operator as follows.

$\varphi\ U_{[t1,t2]}\ \psi$ where φ, ψ are atomic propositions and $U_{[t1,t2]}$ is the strong until operator with time interval [t1, t2]. The Promela codes emulated the strong until operator is shown in Fig. 8.



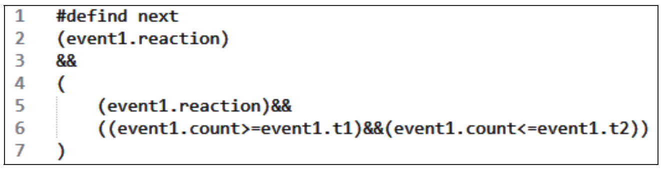
```
#define until
(
    (event1.reaction)
    &&((event1.count>=envent1.t1)&&(event1.count<=event1.t2))
)
&&
(
    ((event2.reaction)
    ||((event2.count>=event2.t1)&&(event2.count<=event2.t2)))
    ||((event1.reaction)
    ||((event1.count>=event1.t1)&&(event1.count<=event1.t2)))
)
```

Fig. 8. The Promela codes which emulate the strong until operator.

2) *Next operation*

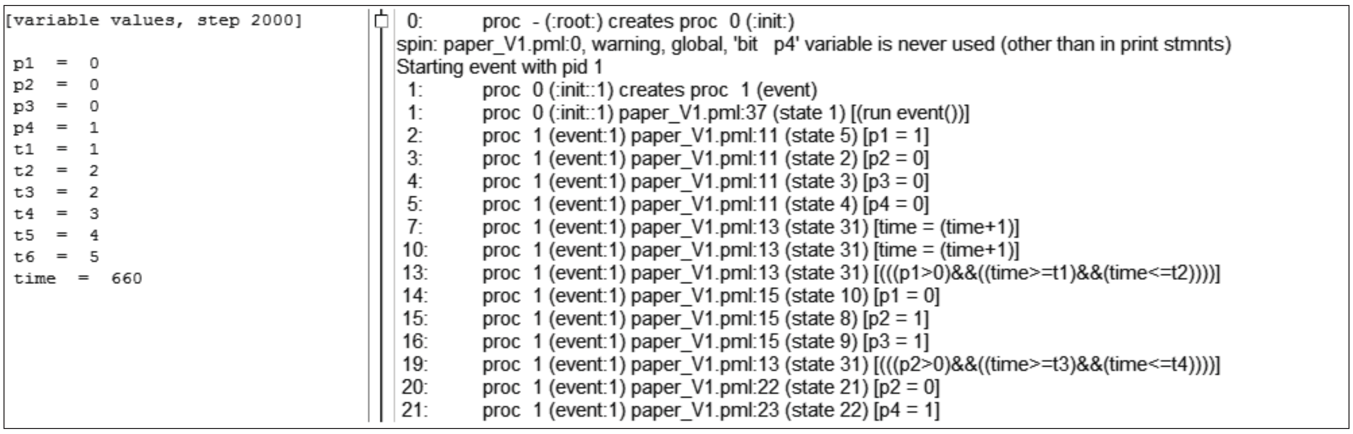
Given a generic formula of the next operator as follows.

$\circ_{[t1,t2]}\varphi$ where φ is an atomic propositions and $\circ$ is the next operator with time interval [t1, t2]. The Promela codes emulated the next operator is shown in Fig. 9.



```
#defind next
(event1.reaction)
&&
(
    (event1.reaction)&&
    ((event1.count>=event1.t1)&&(event1.count<=event1.t2))
)
```

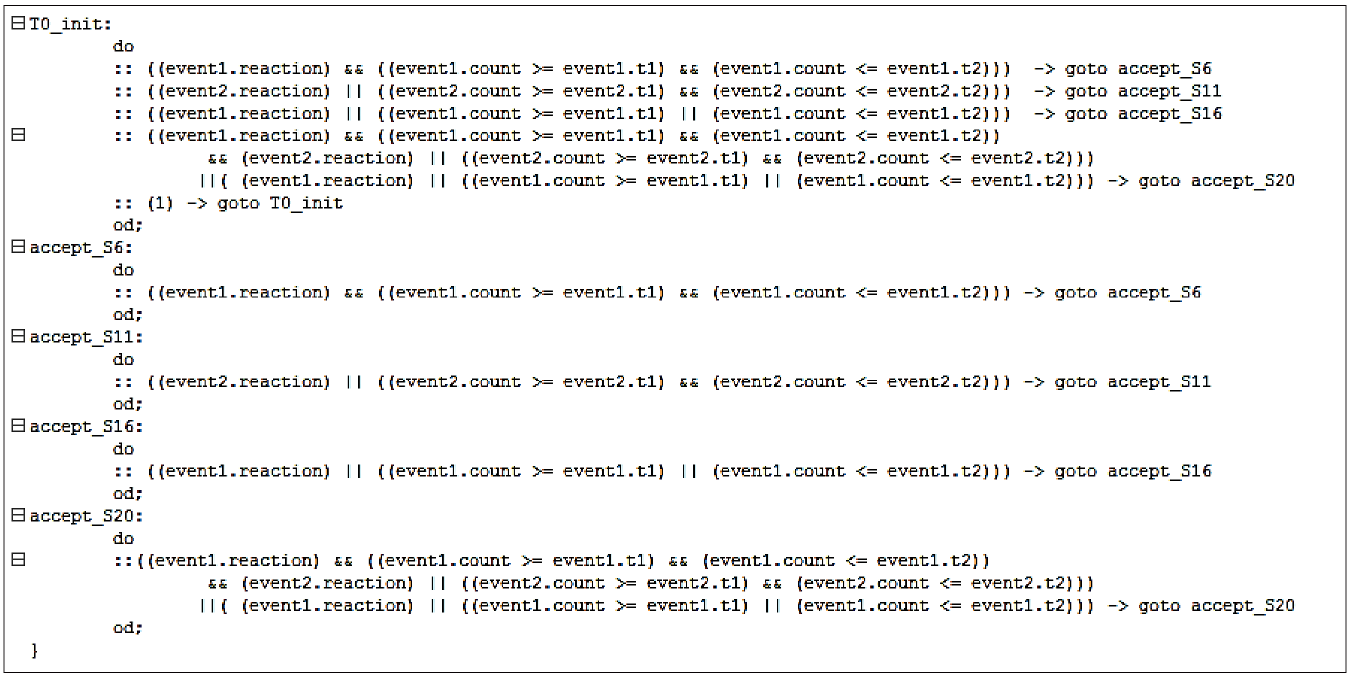
Fig. 9. The Promela codes which emulate the next operator.



```
[variable values, step 2000]

 p1   =   0
 p2   =   0
 p3   =   0
 p4   =   1
 t1   =   1
 t2   =   2
 t3   =   2
 t4   =   3
 t5   =   4
 t6   =   5
 time   =   660
```

```
 0:      proc  - (:root:) creates proc  0 (:init:)
spin: paper_V1.pml:0, warning, global, 'bit   p4' variable is never used (other than in print stmnts)
Starting event with pid 1
 1:      proc  0 (:init::1) creates proc  1 (event)
 1:      proc  0 (:init::1) paper_V1.pml:37 (state 1) [(run event())]
 2:      proc  1 (event:1) paper_V1.pml:11 (state 5) [p1 = 1]
 3:      proc  1 (event:1) paper_V1.pml:11 (state 2) [p2 = 0]
 4:      proc  1 (event:1) paper_V1.pml:11 (state 3) [p3 = 0]
 5:      proc  1 (event:1) paper_V1.pml:11 (state 4) [p4 = 0]
 7:      proc  1 (event:1) paper_V1.pml:13 (state 31) [time = (time+1)]
10:      proc  1 (event:1) paper_V1.pml:13 (state 31) [time = (time+1)]
13:      proc  1 (event:1) paper_V1.pml:13 (state 31) [(((p1>0)&&((time>=t1)&&(time<=t2))))]
14:      proc  1 (event:1) paper_V1.pml:15 (state 10) [p1 = 0]
15:      proc  1 (event:1) paper_V1.pml:15 (state 8) [p2 = 1]
16:      proc  1 (event:1) paper_V1.pml:15 (state 9) [p3 = 1]
19:      proc  1 (event:1) paper_V1.pml:13 (state 31) [(((p2>0)&&((time>=t3)&&(time<=t4))))]
20:      proc  1 (event:1) paper_V1.pml:22 (state 21) [p2 = 0]
21:      proc  1 (event:1) paper_V1.pml:23 (state 22) [p4 = 1]
```

Fig. 10. The result of model system process.



```
T0_init:
        do
        :: ((event1.reaction) && ((event1.count >= event1.t1) && (event1.count <= event1.t2)))  -> goto accept_S6
        :: ((event2.reaction) || ((event2.count >= event2.t1) && (event2.count <= event2.t2)))  -> goto accept_S11
        :: ((event1.reaction) || ((event1.count >= event1.t1) || (event1.count <= event1.t2)))  -> goto accept_S16
        :: ((event1.reaction) && ((event1.count >= event1.t1) && (event1.count <= event1.t2))
                && (event2.reaction) || ((event2.count >= event2.t1) && (event2.count <= event2.t2)))
               ||( (event1.reaction) || ((event1.count >= event1.t1) || (event1.count <= event1.t2))) -> goto accept_S20
        :: (1) -> goto T0_init
        od;
accept_S6:
        do
        :: ((event1.reaction) && ((event1.count >= event1.t1) && (event1.count <= event1.t2))) -> goto accept_S6
        od;
accept_S11:
        do
        :: ((event2.reaction) || ((event2.count >= event2.t1) && (event2.count <= event2.t2))) -> goto accept_S11
        od;
accept_S16:
        do
        :: ((event1.reaction) || ((event1.count >= event1.t1) || (event1.count <= event1.t2))) -> goto accept_S16
        od;
accept_S20:
        do
        ::((event1.reaction) && ((event1.count >= event1.t1) && (event1.count <= event1.t2))
                && (event2.reaction) || ((event2.count >= event2.t1) && (event2.count <= event2.t2)))
               ||( (event1.reaction) || ((event1.count >= event1.t1) || (event1.count <= event1.t2))) -> goto accept_S20
        od;
 }
```

Fig. 11. The result of until accept state.

IV. Demonstration

The Promela codes are consolidated into a final version and simulated using Promela Interpreter version 6.2.7. and iSpin program version 1.1.4. The original model shown in Fig. 5, contains four places and three transitions with the assigned time intervals. The result of the simulation is reported in Fig. 10 which indicates the success of a reachable path of the token from place p1 to the place p4, meaning that the system is terminable. However, in order to do the exhaustive simulation or verification, we use the MTL formula as (p1 == 1) $U_{[t1,\ t2]}$ (P2 == 1 $\wedge$ P3 == 1) which would be emulated by the Promela. By setting the variable named event1 = (p1==1) the variable named event2 = (P2 == 1 $\wedge$ P3 == 1), the result of the acceptance of the verification is shown in Fig. 11.

V. Conclusion

In this paper, we provide the transforming of a time Petri net into Promela and extend the transforming of MTL into Promela in [2] to cover with the next and strong until temporal operators. A case study is demonstrated and the generated Promela codes are shown. However, we still import several Promela template codes from [2] to implement the main clock and sub clocks. The final Promela codes are consolidated and verified using SPIN tool.

References

[1] Sharma, A. *A Refinement Calculus for Promela*. in 2013 International Conference on Engineering of Complex Computer Systems. 2013.

[2] Sukvanich P, Thongtak A, Vatanawood W. *Translating Basic Metric Temporal Logic Formulas into Promela.* Information Science and Applications (ICISA): Springer Science+Business Media Singapore; 2016.

[3] Thati P, Ro¸su G. *Monitoring Algorithms for Metric Temporal Logic Specifications*: Electronic Notes in Theoretical Computer Science 113; 2004.

[4] Ouaknine, J.e. and J. Worrell, *Some Recent Results in Metric Temporal Logic*, in FORMATS. 2008, Springer-Verlag Berlin Heidelberg: France.

[5] Pnueli, A. (1997). *The Temporal Logic of Programs.*, Weizmann Science Press of Israel.

[6] Galton, A., *Temporal Logic*, in Stanford Encyclopedia of Philosophy (SEP). 2010.

[7] Wang, J., *Petri Nets for Dynamic Event-Driven System Modeling.* 2007, CRC Press LLC.

[8] Popova-Zeugmann, L., *Time Petri Nets*. 2013, Springer-Verlag Berlin Heidelberg 2013. DOI=10.1007/978-3-642-41115-1_3

[9] MURATA, T., *Petri nets: Properties, analysis and applications*. 1989, Proceedings of the IEEE. p. 541 - 580.

[10] Boucheneb, H. and R. Hadjidj, *Model Checking of Time Petri Nets, Petri Net, Theory and Applications*. 2008.

[11] BERTHOMIEU, B., F. PERES, and F. VERNADAT., *Modeling and Verification of Real-Time Systems Formalisms and Software Tools* 2008, Great Britain and the United States: John Wiley & Sons, Inc. 386.