Engineering Journal issn: 0125-8281

Article

Automatic Transformation of Ordinary Timed Petri Nets into Event-B for Formal Verification

Chalika Saksupawattanakul[a] **and Wiwat Vatanawood**[b,*]

Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok 10330, Thailand
E-mail: [a]chalika.Sa@student.chula.ac.th, [b]wiwat@chula.ac.th (Corresponding author)

Abstract. The behavioral correctness of real-time software systems relies on both the results of its computation and the clock times when the results are produced. Obviously, formal verification of the safety and correctness of real-time software specification from the very beginning of the software design phase obviously helps us reduce the development efforts. From a practical point of view, the timed Petri net is commonly used to graphically model and illustrate the view of the timed behavior of real-time software systems, which is a good basis for an understanding of a model. However, there is a lack of development process software for the simulation of a timed Petri net. Alternatively, formal verification using the Event-B specification method provides an efficient automatic theorem proving tool which is focused on the development process and provides an efficient verified internal data of software. Unfortunately writing an Event-B specification from scratch is still difficult and a mathematical logic background is needed. In this paper, we propose an automatic transformation of ordinary timed Petri nets into Event-B specifications. The basic notations in the ordinary timed Petri nets are considered and mapped into the code fragments of Event-B. The final resulting Event-B codes are generated in the well-formed format which is required and successfully verified by an Event-B prover called a Rodin tool.

Keywords: Timed Petri nets, Event-B, formal verification, real-time software system.

ENGINEERING JOURNAL Volume 22 Issue 4
Received 19 February 2018
Accepted 15 May 2018
Published 31 July 2018
Online at http://www.engj.org/
DOI:10.4186/ej.2018.22.4.161

DOI:10.4186/ej.2018.22.4.161

1. Introduction

The software industry realizes that the verification of a software design beforehand could yield the reduction of development time and cost. Especially for the real-time and critical software systems, the extra safety criteria should be provided and checked whether the software design satisfies the needs of the user, in addition to its correctness. Recently, formal verification of the real-time software has becomes popular and is thought to be a better choice to perform the exhaustive checking of the timed behaviors, while the real-time software testing is only conducted during the design phase. The formal verification provides a proving process to ensure the targeting of safety properties using the mathematical approaches. Most of the formal verification begins with abstracting the real world system into its abstract representative, called the formal model.

Several formal specification languages have been proposed to describe and fit the variety of formal models. In particular, several researchers proposed the merging of the relevant formal languages to yield new language capabilities. For example, combining the notation Z with a Petri net for model a safety-critical system in [1], even the embedding of the Petri nets formalism into the B abstract system found in [2]. The Petri nets formalism is common to the formal model specifications for the concurrent system, with no time constraints in the first place. The mapping from the Petri net the B language of the B-Method [3] was also proposed to show the possibilities of the incorporation of Petri nets and B language, while the mapping from Event-B into Petri nets was also proposed in a one-way translation manner [4]. In this paper, we will be more specific as to the domain of a real-time system in which the time dependent characteristics are seriously considered. The time constraints of Event-B were proposed in [5],[6],[7] as additional timing properties to Event-B. However, our focus on real-time systems would have to deal with the timed Petri net in particular, not just the Petri net.

Basically, a Petri net [8] is a formal model illustrating a discrete and concurrent system and enabling both qualitative and quantitative analysis of the system's behavior. Several Petri net simulation tools [9] were proposed and developed to check the validity of the Petri nets. When it comes to dealing with the clock times, possibly with a delay, timed Petri nets were proposed [10] instead. The timed Petri nets are easy to use and human-readable, with the graphical symbols and their simulations. In spite of its intuitive model, the simulation approach of Petri nets is still tedious and time consuming, and an exhaustive simulation is only possible with the timed Petri nets. The major limitation of timed Petri nets is the fact that they are still unable to express the internal data of software and do not provide any refined framework for the software development process. As an alternatively, we looked at Event-B, an evolution of B-Method. Event-B is a popular formal method for system-level modeling and analysis, which supports the use of refinement to represent a real-time system. Event-B also provides mathematical proof to verify the expected properties instead of using simulation approach, while several theorem proving tools are developed. We also considered the Rodin tool, which provides a mathematical logic prover and has many plug-in proof frameworks available. However, writing an abstract model in Event-B codes from scratch is difficult and a mathematical logic background is needed. Thus, we intend to propose an automatic transformation of a given ordinary timed Petri net into Event-B codes, so that the real-time software system could be exhaustively verified using a mathematical approach. Several mapping rules are proposed to systematically transform the basic notations of the ordinary timed Petri net into Event-B code fragments, which are finally consolidated into the resulting Event-B machine.

This paper is organized as follows. Section 1 is the introduction and section 2 describes the brief backgrounds of the timed Petri nets and the Event-B. Then Section 3 presents our mapping rules approach for the transformation of ordinary timed Petri nets into Event-B. Section 4 discusses our implementation of the transformation while section 5 discusses the related work; and our conclusion is presented in section 6.

2. Background

In this section, we briefly review the background of several formal objects related to our transformation approach such as the timed Petri nets, Event-B and its support tool, called Rodin.

2.1. Timed Petri Net

In a real-time system, every event or process consumes a certain period of time, no matter how short. The timed Petri net is the extension of the original Petri net to cope with this duration of time for the event or process in a real-time system, which is modelled by a transition symbol in the net. In short, the timed Petri net provides the duration time *d(t)* with each transition *t*, where *d(t)* is equal to, or greater than zero.

In timed Petri net [11], there are two kinds of nodes; transition and place. These are drawn as rectangle and circle symbols, respectively. The nodes of transitions show that the events may occur in a real-time system; while the nodes of places represent the condition for an event to be started. Between transitions and places, there are directed arcs associating between the ordered pairs, in which the possible flows of the net are indicated. There is an arc linking from one pre-place to one transition and also another arc linking from the transition to one post-place. The place preceding the transition is called pre-place, and the place following the transition is called post-place. Arcs between two places or between two transitions are not allowed. There is a set of tokens, with a black dot symbol, drawn on some particular places in the net. The basic notations of the timed Petri net are shown in Fig. 1.

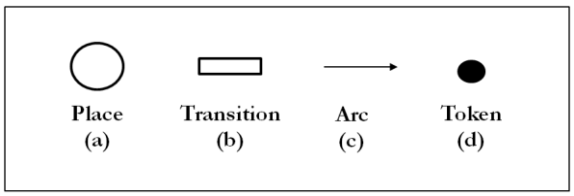

Fig. 1. The basic notations in the timed Petri net [12].

When any transition *t* fires, any token is removed from each of its pre-place and added to any token to each of its post-place. Any distribution of the tokens on the places is called marking. A timed Petri net should be given a specific initial marking to start the flow of the net. However, the firing of the transition *t* is time-dependent in the timed Petri net. When the marking condition of a pre-place is satisfied, the firing of a transition *t* would occur after a certain amount of time (specified duration) *d(t)*. In the graphical representation of a timed Petri net [13], the duration time of each transition is written within an angle bracket next to its symbol, as shown in Fig. 2.

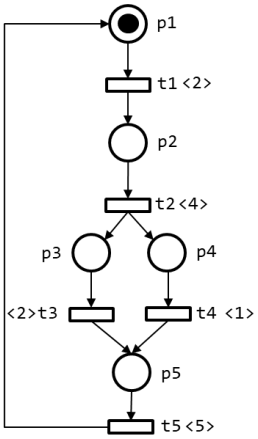

Fig. 2. An example graphic modeling of timed Petri nets.

2.1.1. Our definition of ordinary timed Petri net

In this section, we assume that the given timed Petri net is an ordinary Petri net *OTPN* in which each place holds at most one token. The definitions of our ordinary Petri net are described for the sake of how to understand the element of *OTPN*.

Definition 1: **Ordinary Timed Petri Nets**
A ordinary timed Petri net is a 6-tuple $OTPN = (P, T, F, V, m_0, d)$ such that

- P is a finite set of places
- T is a finite set of transitions
- F is a set of arcs which indicating the flow relations of the net where $F \subseteq (P \times T) \cup (T \times P)$
- V: $F \rightarrow \{1\}$ is weight of the arcs indicating number of token needed to enable the transition firing
- m_0: $P \rightarrow \{0,1\}$ is the initial marking
- d: $T \rightarrow N_0^+$ is the duration time of the transition where $d(t)$ is the non-negative numbers being equal to or greater than the zero of transition t

Definition 2: Pre-place and Post-place Nodes of a Transition
Given an ordinary timed Petri net $OTPN = (P, T, F, V, m_0, d)$. There are two mapping functions, preplace(t) and postplace(t) such that preplace: $T \rightarrow 2^P$ is a function indicating a subset of the pre-place nodes of a transition, while postplace: $T \rightarrow 2^P$ is a function indicating a subset of post-place nodes of a transition according to the existing flows available in *OTPN*.

For example, let (p_1, t_1), (p_2, t_1), $(t_1, p_3) \in F$ where: $p_1, p_2, p_3 \in P$ and $t_1 \in T$. We have preplace$(t_1) = \{p_1, p_2\}$ and postplace$(t_1) = \{p_3\}$.

Definition 3: Marking in an Ordinary Timed Petri Net
Given an ordinary timed Petri net $OTPN = (P, T, F, V, m_0, d)$. A marking in *OTPN* is a total function m: $P \rightarrow \{0, 1\}$. Thus the marking m_0 is a given initial marking to the *OTPN* when the system begins its behavioural actions.

Definition 4: Enabled Transition in an Ordinary Timed Petri Net
Given an ordinary timed Petri net $OTPN = (P, T, F, V, m_0, d)$ and a marking m in *OTPN*. A transition $t \in T$ is enabled in m if m(preplace(t)) = 1. We also define a time counter u: $T \rightarrow N_0^+$ for each transition t which is initially set to zero. Whenever the transition t is enabled, the time counter $u(t)$ begins counting up to $d(t)$.

Definition 5: Firing Transition in an Ordinary Timed Petri Net
Given an ordinary timed Petri net $OTPN = (P, T, F, V, m_0, d)$. The transition $t \in T$ could fire if t is enabled in m_1 and the time counter $u(t) = d(t)$. A firing transition $t \in T$ is denoted by (m_1, t, m_2) where m_1 and m_2 is the markings in *OTPN*. After the firing of t the ordinary timed Petri net *OTPN*, the marking m_1 is removed one token then the marking m_2 is added one token, and the time counter $u(t)$ is reset to zero again. Given an *OTPN* of marking m_1, for each firing transition t with m(preplace(t)) = 1, the m(postplace(t)) is increased by 1 and the m(preplace(t)) is decreased by 1 consecutively to yield a new *OTPN* of marking m_2.

2.2. Event-B

Event-B [14] is a formal framework derived from the original B Method [15] in which it supports the parallel, distributed and reactive system, especially real-time systems. The Event-B formalism also provides a scalable approach to system development, starting from modelling a high level abstract specification of the essential system behaviour and critical properties and continuously doing refinements until achieving sufficiently detailed system specifications. With this Event-B framework, it is possible to verify real-time system properties at the early stages of the system development, by mathematical logic proofs. An Event-B model is mainly written in an Event-B machine with its global context, as shown in Fig. 3. The Event-B machine and its context, called Event-B code in short, usually defines a real-time system in which both structural, and behavioural properties are described in terms of axioms, invariants, events, etc.

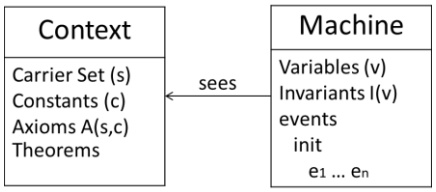

Fig. 3. The structure of an Event-B model [14].

Several formal verification tools are developed to process and prove the Event-B code. The Rodin platform [16] provides support for modelling and verifying Event-B codes with automatic features of proofs. The basic proof obligations are automatically generated by the Rodin tool and proven to ensure the validity of user defined axioms, invariants, and the events along with their guards and actions. Several plug-in provers are available, such as ProB [17], SMT [18], Camille Editor [19], and etc., in order to perform other model checking approaches. An example of Event-B model of a simple traffic lights system is shown in Fig. 4.

```
context SimpleWay
sets LIGHTS DIRECTION
constants Red Green Yellow
axioms
  axm1 partition(LIGHTS; {Red, Green, Yellow})
End

machine SimpleLights sees SimpleWay
variables lights
invariants
   inv1 lights ∈ LIGHTS
events
  event INITIALISATION
  then
   act1 lights ≔ {Red}
  end
```

```
event ToYellow
  any dir
  where
    grd1 lights={Red}
  then
    act1 lights ≔ {Yellow}
 end
event ToGreen
  any dir
  where
    grd1 lights={Yellow}
  then
    act1 lights ≔ {Green}
  end
event ToRed
  any dir
  where
    grd1 lights={Green}
  then
    act1 lights ≔ {Red}
  end
End
```

Fig. 4. An example of Event-B model of the simple traffic lights system [20].

2.2.1. Our definition of Event-B model

The definition of our Event-B model EBM is described for the sake of how to understand the element of EBM.

Definition 6: Event-B Model
An Event-B model is a tuple *EBM = (S, C, A, V, I, E, INIT)* such that

- *S* is a set of model sets or types
- *C* is a set of model constants
- *A* is a set of model axioms
- *V* is a set of variables in model
- *I* is a set of model invariants
- *E* is a set of model events
- *INIT* is the initial action when the model begin its executing

We refer to 'an Event-B model *EBM*' as a convenient way to express our mapping rules. However, each element in an *EBM* should be written in a well-formed statement which is already defined by Event-B code syntax [21]. For example, model sets, model constants, and variables are commonly listed names, while, axioms and invariant are written as a set of logical conditions and the model events are written as a set of any-where-then clauses.

3. Proposed Transformation Scheme

In this section, we propose our transformation scheme to generate Event-B codes from the given timed Petri nets. A set of mapping rules are proposed to generate the Event-B code fragments from timed Petri net notations. We assume that the given timed Petri net is an ordinary timed Petri net in which each place holds at most one token. One of limitation of this paper is that a token holds in each place. It is simple to abstract all its function with set theory. The definitions of our ordinary Petri net and Event-B model are described for the sake of how to understand our transformation mapping rules.

3.1. Our Timed Petri Net to Event-B Model Mapping Rules

The Event-B model is composed of two parts: the static part which is specified by the CONTEXT and the dynamic part which is represented by the MACHINE. Firstly, we apply mapping rules #1 - #4 to transform the structure of the ordinary timed Petri net *OTPN* into the CONTEXT part. While the rest of mapping rules #5 - #7 are applied to transform the behaviors of the *OTPN* consecutively. The resulting CONTEXT and MACHINE parts are written commonly in well-formed statements of Event-B code syntax.

Mapping Rule 1: Transforming Places
Given an ordinary timed Petri net $OTPN = (P, T, F, V, m_0, d)$ and let the target Event-B model be $EBM = (S, C, A, V, I, E, INIT)$, the following steps would be conducted.

1) Add "PLACE" into model sets S, so that "PLACE"$\in S$.
2) Add each p $\in P$ into model constants C, so that p $\in C$.
3) Add each p $\in P$ into model axioms clause as "partition(PLACE, {p} …)" $\in A$.

Mapping Rule 2: Transforming Transitions
Given an ordinary timed Petri net $OTPN = (P, T, F, V, m_0, d)$ and let the target Event-B model be $EBM = (S, C, A, V, I, E, INIT)$, the following steps would be conducted.

1) Add "TRANSITION" into model sets S, so that "TRANSITION" $\in S$.
2) Add each t $\in T$ into model constants C, so that t $\in C$.
3) Add "hold", "enabled", and "firing" into model constants C.
4) Add "PREPL" and "POSTPL" into model constants C.
5) Add each t $\in T$ into model axioms clause as "partition(TRANSITION, {t} …)" $\in A$.
6) Add "STATUS" into model sets S.
7) Add the "partition(STATUS, {hold}, {enabled}, {firing})" clauses in to model axiom A.
8) Add the "PREPL $\in$ TRANSITION $\leftrightarrow$ PLACE" clause into model axiom A.
9) Add the "POSTPL $\in$ TRANSITION $\leftrightarrow$ PLACE" clause into model axiom A.
10) Generate a new set of preplace (t, p) PREPL where t $\in T$, p $\in P$, preplace(t) = p, and add this preplace PREPL into model axioms.
11) Generate a new set of postplace (t, p) POSTPL where t $\in T$, p $\in P$, postplace(t) = p, and add this postplace POSTPL into model axioms.

Mapping Rule 3: Transforming Arcs
Given an ordinary timed Petri net $OTPN = (P, T, F, V, m_0, d)$ and let the target Event-B model be $EBM = (S, C, A, V, I, E, INIT)$, the following steps would be conducted.

1) Generate a new set of inflow (p, t) INFLW where p $\in P$, t $\in T$, (p, t) $\in F$, and add this inflow INFLW into model axioms so that INFLW $\in A$.
2) Generate a new set of outflow (t, p) OUFLW where p $\in P$, t $\in T$, (t, p) $\in F$, and add this outflow OUFLW into model axioms so that OUFLW $\in A$.
3) Add "WEIGHT" into model constants C.
4) Add the "WEIGHT $\in$ N", "WEIGHT = 1" clauses in to model axiom A.

Mapping Rule 4: Transforming Duration Time of Transitions
Given an ordinary timed Petri net *OTPN = (P, T, F, V, m_0, d)* and let the target Event-B model be *EBM = (S, C, A, V, I, E, INIT)*, the following steps would be conducted.

1) Add "durationTime" into model constants *C*.
2) Generate a new set of duration time *d(t)* DTIME, such that its members are denoted as (t, N) where t ∈ *T*, N is the non-negative numbers being equal to or greater than the zero of transition *t*.
3) Add the "durationTime ∈ TRANSITION → N" clause into model axiom *A*.
4) Add this DTIME into model axioms so that DTIME ∈ *A*.

Mapping Rule 5: Transforming Variables and Invariants
Given an ordinary timed Petri net *OTPN = (P, T, F, V, m_0, d)* and let the target Event-B model be *EBM = (S, C, A, V, I, E, INIT)*, we provide a rigid Event-B code fragment of model variables and model invariants as follows.

1) Add "s_PLACE", "s_TRANSITION", "s_MARKING", "STAT", "TIK", and "postPlace" into model variables *V* so that we could handle the behaviours of the net.
2) Add the following invariant clauses into model invariants, "s_PLACE ⊆ PLACE", "s_TRANSITION ⊆ TRANSITION", "s_MARKING ∈ PLACE → Z", "STAT ∈ TRANSITION → STATUS", "TIK ∈ s_TRANSITION → N", and "postPlace ⊆ s_PLACE" so that the all model variables in *V* are defined rigidly.

Mapping Rule 6: Transforming INITIALISATION Events
Given an ordinary timed Petri net *OTPN = (P, T, F, V, m_0, d)* and let the target Event-B model be *EBM = (S, C, A, V, I, E, INIT)*, the following steps would be conducted.

1) Add "s_PLACE := PLACE" and "s_TRANSITION := TRANSITION" clauses into Initial action part *INIT*.
2) Generate a new set of marking (p, q) s_MARKING where p ∈ *P*, q = $m_0(p)$, and add this marking s_MARKING as a clause in *INIT*.
3) Generate a new set of state (t, u) STAT where t ∈ *T*, u ∈ {hold}, and add this state STAT as a clause in *INIT*.
4) Generate a new set of tick (t, 0) TIK where t ∈ *T*, and add this TIK as a clause in *INIT*.
5) Add "postPlace: = ∅" clause into *INIT* so that the postPlace is reset to empty set.

Mapping Rule 7: Transforming Dynamic Events
Given an ordinary timed Petri net *OTPN = (P, T, F, V, m_0, d)* and let the target Event-B model be *EBM = (S, C, A, V, I, E, INIT)*, we propose a rigid Event-B code fragment of model events. We propose to add "Enabled", "Counting", "Fire", "resetIndividualTransition", "resetConcurrentTransition", "removeToken", and "e_DynamicTransition" into model events *E* as follows.

1) The behaviour of any enabled transition *tt* is described with the following "Enabled" event. The guards are defined with two precondition clauses. Firstly, the guard grd1 ensures that any transition *tt* is a member of s_TRANSITION, place *p* is a member of PREPL[{*tt*}] and STAT(*tt*) must hold. Secondly, the guard grd2 ensures that place *p* is a member of s_MARKING, s_MARKING(*p*) has a number of tokens that is equal or greater than WEIGHT and no firing transition occurred, denoted by an empty set of post-Place. Whenever both two precondition guards, grd1 and grd2, are true then a STAT(*t*) is enabled. The "Enabled" event is shown as follows.

"event Enabled
any *p t*
where
grd1 $\exists tt \cdot (tt \in \text{s\_TRANSITION} \wedge (p \in \text{PREPL}[\{tt\}]) \wedge (\text{STAT}(tt) = \text{hold}) \wedge tt = t)$
grd2 $(p \in \text{dom}(\text{s\_MARKING})) \wedge (\text{s\_MARKING}(p) \geq \text{WEIGHT}) \wedge (\text{postPlace} = \emptyset)$
then
act1 $\text{STAT}(t) \coloneqq \text{enabled}$
end"

2) The time counter is described with the following "Counting" event. Similarly, the guards are defined with two precondition clauses. Firstly, the guard grd1 ensures that any transition t is a member of s_TRANSITION and STAT(t) is also enabled. Secondly, the guard grd2 ensures that TIK(t) is less than the duration of time d(t). Whenever both of the two precondition guards, grd1 and grd2, are true then TIK(t) is increased by one. The "Counting" event is shown as follows.

"event Counting
any t
where
grd1 ($t \in$ s_TRANSITION) $\wedge$ (STAT(t) = enabled)
grd2 (TIK(t) < durationTime(t))
then
act1 TIK(t) ≔ TIK(t) +1
end"

3) The behaviour of any fired transition t is described with the following "Fire" event. The guards are defined with two precondition clauses. Firstly, the guard grd1 ensures that any transition t is a member of s_TRANSITION, TIK(t) is equal to the durationTime(t) and STAT(t) is enabled. Secondly, the guard grd2 ensures that place p is a member of s_PLACE, place p is a member of PREPL[{t}] and postPlace[{t}] is a subset of s_PLACE. Whenever both of the two precondition guards, grd1 and grd2, are true then a STAT(t) is firing and POSTPL of transition t is added into the set of postPlace. The "Fire" event is shown as follows.

"event Fire
any t p
where
grd1 ($t \in$ s_TRANSITION) $\wedge$ (TIK(t) = durationTime(t)) $\wedge$ (STAT(t) = enabled)
grd2 ($p \in$ s_PLACE) $\wedge$ ($p \in$ PREPL[{t}]) $\wedge$ (POSTPL[{t}] $\subseteq$ s_PLACE)
then
act1 STAT(t) ≔ firing
act2 postPlace ≔ postPlace $\cup$ POSTPL[{t}]
end"

4) The behaviour of any time counter and status of a transition t would be reset as described with the following "resetIndividualTransition" event. The guards are defined with two precondition clauses. Firstly, the guard grd1 ensures that any transition t is a member of dom(TIK), STAT(t) is firing, a set of postPlace is empty, and POSTPL[{t}] is a subset of s_PLACE. Secondly, the guard grs2 ensures that place p is a member of s_PLACE and place p is a member of PREPL[{t}]. Whenever both of the two precondition guards, grd1 and grd2, are true then a STAT(t) is set to status hold and TIK(t) is reset to zero. The "resetIndividualTransition" event is shown as follows.

"event resetIndividualTransition
any t p
where
grd1 ($t \in$ dom(TIK)) $\wedge$ (STAT(t) = firing) $\wedge$ (postPlace =$\emptyset$) $\wedge$ (POSTPL[{t}] $\subseteq$ s_PLACE)
grd2 ($p \in$ s_PLACE) $\wedge$ ($p \in$ PREPL[{t}])
then
act1 STAT(t) ≔ hold
act2 TIK(t) ≔ 0
end"

5) The behavior of any time counter and status of a transition t would be reset as described with the following "resetConcurrentTransition" event. The guards are defined with two precondition clauses. Firstly, the guard grd1 ensures that any transition t is a member of s_TRANSITION, STAT(t) is also enabled, TIK(t) is not of equal durationTime(t), and place p is a member of PREPL[{t}]. Secondly, the guard grs2 ensures

that place p is a member of s_PLACE and s_MARKING(p) token is zero. Whenever both of the two precondition guards, grd1 and grd2, are true then a STAT(t) is set to status hold and TIK(t) is reset to zero. The "resetConcurrentTransition" event is shown as follows.

"event resetConcurrentTransition
any t p
where
grd1 ($t \in$ s_TRANSITION) $\wedge$ (STAT(t) = enabled) $\wedge$ (TIK(t) $\neq$ durationTime(t))
$\wedge$ ($p \in$ PREPL[{t}])
grd2 ($p \in$ s_PLACE) $\wedge$ (s_MARKING(p) = 0)
then
act1 STAT(t) := hold"
act2 TIK(t) := 0
end"

6) The tokens of the pre-places of any transition t are removed as described with the following "removeToken" event. The guards are defined with two precondition clauses. Firstly, the guard grd1 ensures that *pIn* is a member of the PREPL[{t}], the POSTPL[{t}] is a subset of s_PLACE, and the set of postPlace is an empty set. Secondly, the guard grs2 ensures that *pIn* is a member of s_PLACE, the amount of the number of s_MARKING(*pIn*) is greater than zero, and STAT(t) is firing. Whenever both of the two precondition guards, grd1 and grd2, are true then the number of tokens in s_MARKING(*pIn*) is decreased by WEIGHT. The "removeToken" event is shown as follows.

"event removeToken
any t *pIn*
where
grd1 (*pIn* $\in$ PREPL[{t}]) $\wedge$ (POSTPL[{t}] $\subseteq$ s_PLACE) $\wedge$ (postPlace = $\emptyset$)
grd2 (*pIn* $\in$ s_PLACE) $\wedge$ (s_MARKING(*pIn*) > 0) $\wedge$ (STAT(t) = firing)
then
act1 s_MARKING(*pIn*) := s_MARKING(*pIn*) − WEIGHT
end"

7) The tokens of the post-places of any fired transition t are added as described with the following "e_DynamicTransition" event. The guards are defined with three precondition clauses. Firstly, the guard grd1 ensures that transition t is a member of s_TRANSITION, while *pOut* is a member of s_PLACE, and STAT(t) is firing. Secondly, the guard grs2 ensures that postPlace is not an empty set and *pOut* is a member of postPlace. Thirdly, the guard grs3 ensures that *pIn* is a member of s_PLACE and *pIn* is a member of PREPL[{t}]. Whenever all of the three precondition guards, grd1, grd2, and grd3 are true then the number of tokens in s_MARKING(*pIn*) are increased by WEIGHT and the POSTPL of transition t is removed in the set of postPlace. The "e_DynamicTransition" event is shown as follows.

"event e_DynamicTransition
any t *pIn pOut*
where
grd1 ($t \in$ s_TRANSITION) $\wedge$ (*pOut* $\in$ s_PLACE) $\wedge$ (STAT(t) = firing) $\wedge$ (*pOut* $\in$ POSTPL[{t}])
grd2 (postPlace $\neq \emptyset$) $\wedge$ (*pOut* $\in$ postPlace)
grd3 (*pIn* $\in$ s_PLACE) $\wedge$ (*pIn* $\in$ PREPL[{t}])
then
act1 s_MARKING(*pOut*) := s_MARKING(*pOut*) + WEIGHT
act2 postPlace := postPlace $\setminus$ {*pOut*}
end
end"

4. Implementation

This section describes our propose methodology of transformation which consists of three main steps as shown in Fig. 5. Indeed, an automatic transformation tool is very important to facilitate the generation of Event-B, so we intend to develop the transformation tool within an Eclipse environment using Java language. Firstly, we expect an input of ordinary timed Petri net *OTPN* written in terms of the well-formed structure of the XML file format. The notations of *OTPN* are thoroughly extracted. Secondly, our proposed mapping rules are exploited to generate the corresponding Event-B model *EBM*, including CONTEXT part and MACHINE part as to represent both structural and behavioural properties of *OTPN* respectively. The Event-B code fragments are consolidated and elaborated to comply with the Event-B code syntax, so that the resulting final model would be complete and ready to the next verification step. Thirdly, the resulting Event-B model would be evaluated for its correctness and consistency, using the relevant proof obligations. The proof obligations would be determined from the original properties of *OTPN*, such as liveness, safetyness, and even the fairness properties written in linear temporal logic formula.

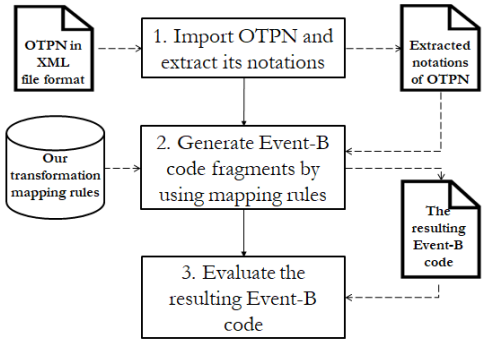

Fig. 5. Our proposed methodology of transformation.

4.1. Import *OTPN* and Extract Its Notations

In order to demonstrate how to import the input *OTPN*, we designed a simple and readable XML schema with only three main elements: <Places> indicating a place, <Transitions> indicating a transition, and <Arcs> which represents an arc. The <Places> contains the attributes for the place *id, name, marking* (number of carrying token), and *type*. The <Transitions> contains the attributes for the transition *id, name, duration time*, and *type*. While, the <Arcs> contains the attributes for the arc *id, type, fromNode,* and *toNode*. An example of the XML file of the *OTPN* is shown in Fig. 6. According to the definition of the ordinary timed Petri net, it is common to verify the well-formedness of the input XML of an *OTPN* and promptly alert the error found beforehand. The XML form of *OTPN* is more practically traced to extract the collection of the places, transitions, and arcs with the adequate information in order to apply the mapping rules in the next step. To illustrate the result of this step, the collections of notations are shown in Fig. 7 regarding the definition of $OTPN = (P, T, F, V, m_0, d)$.

```
<?xml version="1.0" encoding="UTF-8"?>
<OTPNs>
    <Places>
        <place id="PL_1" name= "p1" initialMarking="1" type="node"/>
        <place id="PL_2" name= "p2" initialMarking="0" type="node"/>
        <place id="PL_3" name= "p3" initialMarking="0" type="node">
    </Places>
    <Transitions>
        <Transition id="TR_1" name= "t1" durationTime="2" type="node"/>
        <Transition id="TR_2" name= "t2" durationTime="4" type="node"/>
    </Transitions>
  <Arcs>
        <Arc id="in_1" type="inputPlace" fromNode="PL_1" toNode="TR_1"/>
        <Arc id="in_2" type="inputPlace" fromNode="PL_2" toNode="TR_2"/>
        <Arc id="out_1" type="outputPlace" fromNode="TR_1" toNode="PL_2"/>
        <Arc id="out_2" type="outputPlace" fromNode="TR_2" toNode="PL_3"/>
  </Arcs>
</OTPNs>
```

Fig. 6. An example of the XML file of the *OTPN*.

Element Type	Ordinary timed Petri net *OTPN =(P, T, F, V, m_0, d)*
<Places> <place id="PL_1" name="p1" initialMarking="1" type="node"/> <place id="PL_2" name="p2" initialMarking="0" type="node"/> <place id="PL_3" name="p3" initialMarking="0" type="node"/> </Places>	P = {p1,p2,p3} m_0 = (1,0,0)
<Transitions> <Transition id="TR_1" name="t1" durationTime="2" type="node"/> <Transition id="TR_2" name="t2" durationTime="4" type="node"/> </Transitions>	T = {t1,t2} D = {(t1,2),(t2,4)}
<Arcs> <Arc id="in_1" type="inputPlace" fromNode="PL_1" toNode ="TR_1"/> <Arc id="in_2" type="inputPlace" fromNode="PL_2" toNode ="TR_2"/> <Arc id="out_1" type="outputPlace" fromNode="TR_1" toNode ="PL_2"/> <Arc id="out_2" type="outputPlace" fromNode="TR_2" toNode ="PL_3"/> </Arcs>	F = {(p1,t1),(p2,t2), (t1,p2),(t2,p3)} V = {(p1,t1)→1,(p2,t2)→1, (t1,p2)→1,(t2,p3)→1}

Fig. 7. An example of the resulting collections of notations found in the *OTPN*.

4.2. Generate Event-B Code Fragments by Using Mapping Rules

According to a sample of an ordinary timed Petri net *OTPN* in Fig. 2, the collections of five places and five transitions with defined duration times or delay times, and one initial marking are extracted. In this step, our mapping rules are exploited to generate the corresponding Event-B model written in CONTEXT and MACHINE parts. As we mentioned earlier, the CONTEXT part describes the structure of the *OTPN* and the MACHINE part describes the behaviours of the *OTPN*. We apply mapping rules # 1 - #4 to generate the model sets, model constants, and model axioms of the CONTEXT from the places, transitions, arcs, and duration time of *OTPN*, as shown in Fig. 8.

#	Event-B code
1	context c_TransformationOTPN2EBM
2	sets PLACE TRANSITION STATUS
3	constants p1 p2 p3 p4 p5 t1 t2 t3 t4 t5 hold enabled firing delay weight pre post
4	axioms
5	axm1 partition(PLACE,{p1},{p2},{p3},{p4},{p5})
6	axm2 partition(TRANSITION,{t1},{t2},{t3},{t4},{t5})
7	axm3 partition(STATUS,{hold},{enabled},{firing})
8	amx4 PREPL ∈ PLACE ⇸ TRANSITION
9	axm5 PREPL = {p1↦t1,p2↦t2,p3↦t3,p4↦t4,p5↦t5}
10	axm6 POSTPL ∈ TRANSITION ⇸ PLACE
11	axm7 POSTPL = {t1↦p2,t2↦p3,t2↦p4,t3↦p5,t4↦p5,t5↦p1}
12	axm8 weight ∈ ℕ
13	axm9 weight = 1
14	axm10 delay ∈ TRANSITION → ℕ
15	axm11 delay = {t1↦2,t2↦4,t3↦2,t4↦1,t5↦5}
16	End

Fig. 8. An example of the generated CONTEXT part of Event-B model.

Afterward, the rest of the mapping rules #5 - #7 are used to generate the variables, INITIALISATION event, and several rigid events of "Enabled", "Counting", "Fire", "resetIndividualTransition", "resetConcurrentTransition", "removeToken", and "e_DynamicTransition" for MACHINE part from the semantic of the *OTPN*, as shown in Fig. 9.

#	Event-B code
1	machine m_TransformationTPN2EB sees c_TransformationTPN2EB
2	variables s_PLACE s_TRANSITION s_MARKING STAT TIK postPlace
3	invariants
4	inv1 s_PLACE ⊆ PLACE
5	inv2 s_TRANSITION ⊆ TRANSITION
6	inv3 s_MARKING ∈ s_PLACE → ℤ
7	inv4 STAT ∈ TRANSITION → STATUS
8	inv5 TIK ∈ s_TRANSITION → ℕ
9	inv6 postPlace ⊆ s_PLACE
10	events
11	event INITIALISATION
12	then
13	act1 s_PLACE ≔ PLACE
14	act2 s_TRANSITION ≔ TRANSITION
15	act3 s_MARKING ≔ {p1 ↦ 1, p2 ↦ 0, p3 ↦ 0 , p4 ↦ 0,p5 ↦ 0}
16	act4 STAT ≔ {t1 ↦ hold,t2 ↦ hold,t3 ↦ hold ,t4 ↦ hold,t5 ↦ hold}
17	act5 TIK ≔ {t1 ↦ 0, t2 ↦ 0, t3 ↦ 0, t4 ↦ 0, t5 ↦ 0}
18	act6 postPlace ≔ ∅
19	end
20	event Enabled
21	any p t
22	where
23	grd1 ∃tt·(tt ∈ s_TRANSITION ∧ (p∈ PREPL[{tt}]) ∧ (STAT(tt) = hold) ∧ tt = t)
24	grd2 (p ∈ dom(s_MARKING)) ∧ (s_MARKING(p) ≥ WEIGHT) ∧ (postPlace = ∅)
25	then
26	act1 STAT(t) ≔ enabled
27	end
28	event Counting
29	any t
30	where
31	grd1 (t ∈ s_TRANSITION) ∧ (STAT(t) = enabled)
32	grd2 (TIK(t) < durationTime(t))
33	then
34	act1 TIK(t) ≔ TIK(t) +1
35	end
36	event Fire
37	any t p
38	where
39	grd1 (t∈s_TRANSITION) ∧ (TIK(t) = durationTime(t)) ∧ (STAT(t) = enabled)
40	grd2 (p ∈ s_PLACE) ∧ (p ∈ PREPL[{t}]) ∧ (POSTPL[{t}] ⊆ s_PLACE)
41	then
42	act1 STAT(t) ≔ firing
43	act2 postPlace ≔ postPlaceU POSTPL[{t}]
44	end

#	Event-B code
45	event resetIndividualTransition
46	any t p
47	where
48	grd1 (t∈ dom(TIK)) ∧ (STAT(t) = firing) ∧ (postPlace = ∅) ∧ (POSTPL[{t}]⊆s_PLACE)
49	grd2 (p ∈ s_PLACE) ∧ (p ∈ PREPL[{t}])
50	then
51	act1 STAT(t) ≔ hold
52	act2 TIK(t) ≔ 0
53	end
54	event resetConcurrentTransition
55	any t p
56	where
57	grd1 (t∈s_TRANSITION) ∧ (STAT(t) = enabled) ∧ (TIK(t) ≠ durationTime(t))∧ (p ∈PREPL[{t}])
58	grd2 (p ∈ s_PLACE) ∧ (s_MARKING(p) = 0)
59	then
60	act1 STAT(t) ≔ hold
61	act2 TIK(t) ≔ 0
62	end
63	event removeToken
64	any t pIn
65	where
66	grd1 (pIn ∈ PREPL[{t}]) ∧ (POSTPL[{t}] ⊆ s_PLACE) ∧ (postPlace = ∅)
67	grd2 (pIn ∈ s_PLACE) ∧ (s_MARKING(pIn) > 0) ∧(STAT(t) = firing)
68	then
69	act1 s_MARKING(pIn) ≔ s_MARKING(pIn) − WEIGHT
70	end
71	event e_DynamicTransition
72	any t pIn pOut
73	where
74	grd1 (t∈s_TRANSITION) ∧ (pOut ∈ s_PLACE) ∧ (STAT(t) = firing) ∧ (pOut ∈ POSTPL[{t}])
75	grd2 (postPlace ≠ ∅) ∧ (pOut ∈ postPlace)
76	grd3 (pIn ∈ s_PLACE) ∧ (pIn ∈ PREPL[{t}])
77	then
78	act1 s_MARKING(pOut) ≔ s_MARKING(pOut) + WEIGHT
79	act2 postPlace ≔ postPlace \ {pOut}
80	end
81	end

Fig. 9. An example of the generated MACHINE part of Event-B model.

4.3. Evaluate the Resulting Event-B Code

In this section, we demonstrate the evaluation of the resulting Event-B code in two steps. Firstly, the Event-B code would be verified for its correctness and consistency among every single statement of the source code by using the proof obligations. With the Rodin tool, this verification step would be commonly done. Secondly, we could selectively verify some essential safety properties of the system using an open source plug-in to the Rodin tool, called the ProB plug-in.

4.3.1. Verify the correctness and consistency of the Event-B code

Using the Rodin tool, the mathematical proof is automatically possible. The proof statistics of the resulting Event-B code are in Table 1. All the proof obligations (POs) for the eight events and six invariants generated

earlier are successfully proved by the Rodin prover and the proof statistics are listed. The total results of the resulting Event-B code is 41 POs, within which 41 POs (100%) are proved automatically by the Rodin prover and if there is no interactively discharge that is not proven automatically, a user interface allows interactive proof steps.

Table 1. Proof statistics of our generated Event-B code.

Element Type	Element Name	Number of proof obligations	Automatically discharged	Interactively discharged	Percentage %
Event	Enabled	3	3	0	100
	Counting	4	4	0	100
	Fire	3	3	0	100
	INITIALISATION	3	3	0	100
	resetIndividualTransition	3	3	0	100
	resetConcurrentTransition	4	4	0	100
	removeToken	3	3	0	100
	e_DynamicTransition	4	4	0	100
Invariant	inv1	0	0	0	100
	inv2	0	0	0	100
	inv3	3	3	0	100
	inv4	5	5	0	100
	inv5	4	4	0	100
	inv6	2	2	0	100
	Total	**41**	**41**	**0**	**100%**

4.3.2. Verify the essential safety properties of Event-B model

The safety property ensures that bad or unwanted things ever happen. The essential safety properties are the invariants of the system. In short, we would ensure that all invariants defined should always hold. In Fig. 10, the result of the Rodin tool shows that no violation of the invariants of the system written in Event-B code appears.

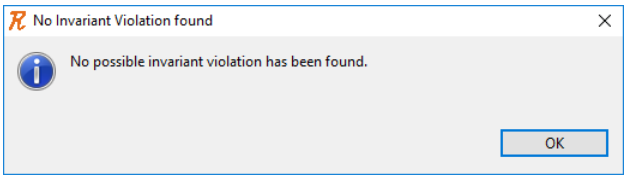

Fig. 10. The violation of the system invariants is not found.

Moreover, temporal logic formula (LTL) has found an important application in formal verification, where it is used to verify propositions qualified in terms of time of hardware or software systems. Consequently, we show additional safety properties written in linear temporal logic LTL formula regarding the safety of the local clocks of each transition, "The local clocks of the transitions never count beyond their assigned duration times". The LTL clause is shown as follows.

$$F\ \{TIK(t_1) <= durationTime(t_1)\ \&\ TIK(t_2) <= durationTime(t_2)\ \&\ TIK(t_3) <= durationTime(t_3)\ \&\ TIK(t_4) <= durationTime(t_4)\ \&\ TIK(t_5) <= durationTime(t_5)\}.$$

The *F* is the temporal operator called "Finally". While *TIK(t)* is the counting of local clock of transition *t* and *durationTime(t)* is the duration time assigned to transition *t*.

5. Related Work

There have also been researchers that combined Event-B with other formal methods for the specification of systems, primarily with the Petri net. The work [2] proposes the embedding techniques for embedding of Petri nets formalisms into the B abstract. . They outline the embedding, which enables one to conjointly use Petri nets and Event-B in the same system development. The essential difference between our approach and this work is that we translate ordinary timed Petri net to the Event-B model, so that each behavior is tied to a separate event. This is more natural and usable for real-time systems. They use the Atelier B version of the Event-B syntax, which is much closer to the classical B-language. However, the Atelier B prover is usually more difficult to use in our approach. In [3] we propose a method of mapping PNs to B-language that is useful for the incorporation of Petri net designs in a software application developed by the B-Method. However, the B-Method was designed for the development of sequential systems, so it is limited when used to check concurrent systems. But, while there is considered to be a strong relation between the computational concepts of Petri nets and Event-B; it will also be worthwhile to explore the possibilities of incorporation of transformed PN into Event-B models. The work [7] proposes a formal model of timed migrating and communicating process as provided by the TiMo calculus. They translate the specifications of TiMo to Event-B, which transfers additional timing property to Event-B. They analyze the software systems by using the Rodin tool for the theorem proving and model checking techniques. However, they are not aware of Event-B timing constraints; they implement local clock and relative time suites instead of global clock and absolute time. The case study in [22] is similar. Transformation is used in a railway safety-related. The authors translate a Colored Petri net CPN specification to the Event-B language by using the Altelier syntax. The B specification uses a special machine that implements multi-sets and the purpose of this transformation is a further development of the specified system.

6. Conclusion

This paper proposes an alternative to just automatically generating the Event-B model *EBM* from a given ordinary timed Petri net *OTPN*. Typically, it is common to abstract a real-time system using *OTPN* net, and the net simulation is the only the practical way to evaluate, and verify its critical properties. While the simulation technique is tedious and time consuming to conduct for the verification of a real-time system, the theorem proving and model checking are considered. We believe that since the Event-B model and its model prover tools have recently become popular for critical system development, new ideas are required. We propose a set of mapping rules of automatic transforming a commonly used *OTPN* net into Event-B model, without a mathematical background for writing an Event-B specification. The time dependent mechanism has been exploited for the Event-B model using the counting of local clocks for every single transition found in *OTPN*. The structural and behavioral parts of the *OTPN* have been transformed into the CONTEXT part and MACHINE part of Event-B codes respectively. The resulting Event-B code is syntactically well-formed. Moreover, the proof of its correctness, consistency and essential safety properties are demonstrated by using the Rodin tool with the additional plug-in called ProB. As future work, we plan to cover in our Event-B model some of the ordinary timed Petri net as the additional priority of a transition fired as define in [11]. This extension became necessary in order to support the specification of priority-based schedules as used in real-time systems. This can be integrated into our algorithm of mapping rules as an extra refinement.

References

[1] M. H. Monika Heiner, "Modeling safety-critical systems with Z and Petri nets," in *SAFECOMP*, Berlin, Heidelberg, 1999, pp. 361-374.

[2] J. C. Attiogbé, "Semantic Embedding of Petri Nets into Event-B," in *International IM_FMT*, Dusseldorf, 2009.

[3] S. Korecko, "Petri Nets to B-language transformation in software development," *Acta Polytechnica Hungarica*, vol. 11, no. 6, 2014.

[4] B. M. K. Mohamed Garoui, "The EventB2PN Tool: From Event-B specification to Petri Nets through model transformation," in *015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, Takamatsu, 2015.

[5] D. M. R. Dominique Cansell, "Time constraint patterns for Event B development," in *International Conference of B Users*, Berlin, 2006.
[6] M. W. L. T. Faezeh Siavashi, "Modeling critical systems with timing constraints in Event-B," in *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2013, vol. 4, pp. 70-72.
[7] T. S. H. S. Gabriel Ciobanu, "From TiMo to Event-B: Event-driven timed mobility," in *2014 19th International Conference on Engineering of Complex Computer Systems*, Tianjin, China, 2014.
[8] B. W. Choi, "Petri net approaches for modelling, controlling and validating flexible manufacturing systems," Ph.D. Dissertations, no. 10687, Industrial and Manufacturing Systems Engineering, Iowa State University Capestones, 1994.
[9] *Petri Nets Tools Database Quick Overview* [Online]. Available: https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html
[10] C. Ramchandani, "Analysis of asynchronous concurrent systems by Timed Petri nets," Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.
[11] L. Popova-Zeugmann, *Time and Petri Nets*. Berlin, Germany: Springer, 2013.
[12] G. Callou, P. Maciel, D. Tutsch, J. Araújo, J. Ferreira and R. Souza, "A Petri Net-based approach to the quantification of data center dependability," in *Petri Nets - Manufacturing and Computer Science*. InTech, 2012.
[13] M. Werner, L. Popova-Zeugmann, and J. Richling, "A method to prove non-reachability in priority duration Petri nets," *Fundamenta Informaticae,* vol. 61, no. 3-4, pp. 351-368, 2004.
[14] J.-R. Abria, *Modeling in Event-B System and Software Engineer*. Cambridge University Press, 2010.
[15] J.-R. Abrial, *The B-Book—Assigning Programs to Meanings* Cambridge University Press, 2005.
[16] M. J. M. B. Covers Rodin, *Rodin User's Handbook*, M. Jastram, Ed. 2012.
[17] *The ProB Animator and Model Checker*, Heinrich-Heine-University, Institut für Software und Programmiersprachen. [Online]. Available: https://www3.hhu.de/stups/prob/index.php/Main_Page
[18] P. F. G. L. V. David Déharbe, "SMT solvers for Rodin," in *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, Springer Berlin Heidelberg, 2012, pp. 194-207.
[19] University of Düsseldorf, *Camille Editor* [Online]. Available: http://wiki.event-b.org/index.php/Camille_Editor
[20] K. Robinson, "Refinement," in *System Modelling & Design Using Event-B*. University of New South Wales, 2010.
[21] D. Cansell and D. Méry, "Tutorial on the event-based B method," Paris, 2006.
[22] T. Kiss and K. T. Janosi-Rancz, "Developing railway interlocking systems with session types and Event-B," in *2016 IEEE 11th Int. Symp. Appl. Comput. Intell. Informatics*, 2016, pp. 93–98.