國立交通大學

資訊科學與工程研究所

博 士 論 文

運用一具 Hierarchy 特性的 Timed CPNets 技術來分析 BPMN 工作流程

Applying Timed CPNets with Hierarchy to Analyze a Workflow in BPMN

研 究 生：王靜慧

指導教授：王豐堅 教授

中 華 民 國 九 十 八 年 七 月

運用一具 Hierarchy 特性的 Timed CPNets 技術來分析 BPMN 工作流程

Applying Timed CPNets with Hierarchy to Analyze a Workflow in BPMN

研 究 生：王靜慧　　Student：Ching-Huey Wang

指導教授：王豐堅　　Advisor：Feng-Jian Wang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

博 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Doctor

in

Computer Science

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年七月

運用一具 Hierarchy 特性的 Time CPNets 技術來分析 BPMN 工作流程

學生：王靜慧　　　　　　　　　　　　　指導教授：王豐堅 博士

國立交通大學資訊工程與科學研究所 博士班

摘 要

相對於 BPMN 而言，現存的商業流程之研究或商業軟體大多只提供或使用其中的部分。BPMN 主要的組成元素包括:控制流程、訊息流程、資料流程以及角色分配。它也提供多實體 activity、事件觸發 activity 及進階控制機制。雖然這些元素讓 BPMN 具更大的流程表達能力，但也增加了設計階段其所表達之流程的分析困難度。本論文提出一個正規流程模型來協助根據 BPMN 四種組成元素所描述的商業流程。同時，也提供一具階層特質之時間顏色派翠網模型。並建立一套流程與此網的轉換規則，以便將上述 BPMN 商業流程轉換成相對之時間顏色派翠網，來運用既有之分析方法做靜態分析—如 deadlock 檢查。在本論文中，我們更進一步探討 well-formed 和 unstructured 相當普遍的流程之分析。此外，我們將以一個實際的例子做示範，利用時間顏色派翠網 deadlock 分析方法，再根據其結果推斷可能會影響流程執行的異常 artifact 之使用。最後，我們也將比較刻下技術與我們之研究成果。

關鍵字: 商業流程模型符號，工作流程，商業流程，分析，控制流程，資料流程，訊息流程、顏色派翠網、時間顏色派翠網、階層式派翠網

Applying Timed CPNets with Hierarchy to Analyze a Workflow in BPMN

Student：Ching-Huey Wang　　　　Advisor：Dr. Feng-Jian Wang

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Although many business process models have been proposed, most of them do not apply all the following arguments: *control*, *message* and *data flows* and *role assignments*, defined completely in BPMN. Besides, they do not provide the multi-instance activity, event-triggered activity or the control node with complex mechanisms as in BPMN. On the other hand, these features allow a process to be defined with richer semantics but increase the difficulty of correcting an error or inaccurate process in workflow design.

This thesis proposes a formal process model to help describing a BPMN-based process. To simplify the analysis, we also provided Hierarchical Timed Coloured Petri Nets (H_C^T PNets), which is extended from Time Coloured Petri Nets with hierarchy and allows some analysis with existing techniques. Once a workflow based on our BPMN model is specified, a series of mapping rules can be used to transform the workflow into a H_C^T PNets for analysis. An example is applied to demonstrate the transformation and the corresponding deadlock detection. Furthermore, the artifact usage anomaly detection mechanisms within either a well-formed or unstructured process are discussed. Finally, a comparison among related works and ours and the future works are presented.

Keyword: BPMN, workflow, business process, analysis, control flow, data flow, message flow, CPNets, Time CPNets and hierarchical PNets.

誌 謝

本篇論文的完成，首先要萬分感謝指導教授王豐堅博士，王教授在我求學期間持續不斷的指導與鼓勵，讓我不僅在論文研究方面學習到相當寶貴的經驗，在做人處事方面也獲益良多。如今學生若有些微的成就，王教授的指導實在功不可沒。

其次要感謝吳毅成教授，陳耀宗教授，朱治平教授，黃冠寰教授，焦惠津教授，梅興教授，與楊鎮華教授，在百忙之中首肯擔任我博士論文的口試委員，並且提供了許多寶貴的意見，補足我論文裡不足的部分。其中吳、陳兩位教授亦是我的論文計畫書口試委員，在論文報告的方式上也給了我相當多的指導。此外，對於一起研究討論與互相鼓勵的實驗室學長與學弟們，包括楊基載、黃國展、王建偉、許嘉麟、許懷中、黃培書等等，在此一併加以感謝。

最後，我要與我的家人、同學、朋友共同分享完成論文的喜悅，由於有您們的支持與關懷，陪伴我走過這漫長的求學過程。僅將此論文獻給我最敬愛的父母親與支持我的親友們。

Table of Contents

List of Figures and Tables

Chapter 1. Introduction

Workflow can be viewed as a set of interrelated tasks that are systematized to achieve certain business goals by completing the tasks in a particular order under automatic control [1]. The Business Process Modeling Notation (BPMN) [2] is a standard for capturing workflow in the early phases of system development. Existing researches focus on 1) parts of the concepts included in BPMN only, e.g., control flow analysis [3][48] or 2) how to transform from control and message flow in BPMN into BPEL code [4][5][6].

A BPMN-based workflow is described with four entities: 1) role: describing the performers of task instantiated, 2): control flow: defining what, when and how tasks a workflow performs, 3) data flow: specifying what information entities are produced/manipulated/passed in corresponding activities and 4) message flow: representing the interaction between processes through messages. An analysis based on the correlations among these four entities can help check or maintain consistency between execution order and data transition [7][8][9][10], as well as prevents exceptions due to contradiction between data flow, control and message interaction.

There are five additional features introduced in BPMN, but not included in traditional process modeling languages. These features allow defining: 1) an interaction between participants, 2) a multi-instance (loop) activity 3) an event-triggered (supplement) process, 4) a join node designed by one of the three advanced join mechanisms, *discriminator*, *multiple merge* and *N out of M join*, and 5) a data flow described with explicit channel. In addition, time event-triggered behaviors can be described in a BPMN-based workflow, i.e., time constrains are embedded. These features allow defining a process with richer semantics, but increase the difficulty of identifying the problems such as inaccuracy in a process specification at design time.

Here, we provide an easier way to extract knowledge from the four entities of a workflow. Based on our previous work [11], a method for describing a BPMN-based process is proposed. Then, we propose a model, Hierarchical Timed Coloured Petri Nets (H_C^T PNets), extended from Timed Coloured Petri Nets (TCPNets) with hierarchy [13][14] for analysis. There are a series of mapping rules defined to transform a BPMN-based process into H_C^T PNets, in which a set of analysis techniques works [14].

With our methodology, the artifact usage anomalies in our previous work are refined. An analysis method of control, data, and message flow is derived. An example is used to indicate our contribution of process development and anomaly detections. Finally, a comparison among ours and related works is presented.

The remainder of this paper is organized as follows. Chapter 2 introduces the Petri Nets and its extensions, Coloured Petri Nets (CPNets), and TCPNets. It also compares existing flow specification model and BPMN. Besides, H_C^T *PNets* is proposed for the problems identified. Chapter 3 presents our business process model, including the control flow, data flow and message flow. In Chapter 4, we present a set of rules transforming a process in BPMN into H_C^T *PNets*. In Chapter 5, the well-behaved unstructured processes are identified and formulated. In Chapter 6, we present a case to demonstrate our methodologies including development and analysis. A comparison between our approach and related works on BPMN is given in Chapter 7. Finally, a conclusion and some recommendations of future works are given in Chapter 8.

Chapter 2. Petri Nets – PNets

PNets, Petri Nets, is a formal model with graphical representations. The original PNets was developed by Petri [27], and various extensions have been developed with their own constructs. Some of these extensions are associated with easier modeling mechanism and keep the same expressiveness as classical PNets [28] and some provide more expressional power [22][23]. PNets has been applied to many areas, including workflow applications [29][30][31]. In this chapter, we discuss the problems rising when applying PNets or its extensions, *Coloured Petri Nets* and *Time Petri Nets*, to analyze business processes represented with BPMN. Before the discussion, definitions of PNets and the two extensions are given.

2.1. Definition of Classical Petri Nets

A PNet, defined in Definition 2.1, is a directed graph with two kinds of nodes, named *place* and *transition*. In general, a place is presented with a circle while a transition is presented with a rectangle. There are no arcs connecting two places or two transitions. An example of PNet is shown in Figure 2.1 where there are three places, two transitions and one token.

Definition 2.1 (Classical Petri Nets – PNets)

A Petri net is a 4-tuple $PNet = (P, T, F, m_0)$ where

1. P is a finite set of places,
2. T is a finite set of transitions such that $P \cap T = \phi$,
3. F is a finite set of directed arcs, $F \subseteq (P \cup T) \times (P \cup T)$, satisfying

$F \cap (P \times P) = F \cap (T \times T) = \phi$,

4. m_0 is the initial marking function, $m_0 : P \to \mathbb{N}$ where $\mathbb{N} = \{1,2,...\}$.

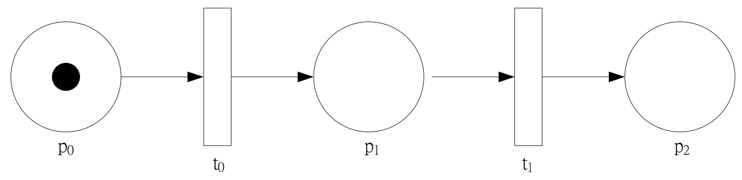


Figure 2.1 An example of a PNet.

Definition 2.2 (Marking)

1. A *marking* M of a set of places P is a mapping $m : P \to \mathbb{N}$ where $\mathbb{N} = \{0,1,2,...\}$.

2. A *marking* M of a Petri net $PNet = (P,T,F,m_0)$ is a marking of P. Initial marking M_0 of $PNet$ is generated by function m_0.

In Definition 2.2, function m is defined from a place to a nonnegative integer which means the number of tokens on the place. A PNet is also equipped with an *initial marking* M_0, i.e., an initial state of the PNet is associated with one or more token in some place(s). All the states of this net succeed to M_0, generated by function m_0. Marking M_0 of an example PNet shown in Figure 2.1 can be expressed as an array based on the order (p_0, p_1, p_2) with nonnegative integers $(1,0,0)$.

Definition 2.3 (Input/Output Set)

Let $PNet = (P,T,F,m_0)$ be a Petri net, for an element $x \in P \cup T$

1. its *input set* ${}^{\bullet}x$ is defined as ${}^{\bullet}x=\{y\in P\cup T\mid(y,x)\in F\}$ and

2. its *output set* $x^{\bullet}$ is defined as $x^{\bullet}=\{y\in P\cup T\mid(x,y)\in F\}$.

Definition 2.3 defines the notations about the input and output sets of a node (place or transition) in a PNet. Note that both sets of a place contain transitions only and both sets of a transition contain places only.

Definition 2.4 (Fire a Transition Enabled)

A transition t is able to be fired (named as enabled) if $\forall p\in{}^{\bullet}t$, $m(p)\geq 1$. Firing t transforms marking M into marking M' and the transformation can be defined from place p by function m and m' as

$$m'(p)=\begin{cases} m(p)-1 & if \quad p\in{}^{\bullet}t-t^{\bullet}, \\ m(p)+1 & if \quad p\in t^{\bullet}-{}^{\bullet}t, \\ m(p) & otherwise. \end{cases}$$

When t is enabled in M, t may fire to change marking M to another marking M'. The new marking M' is obtained by removing one token from each of its input places ${}^{\bullet}t$ and by putting one token to each of its output places $t^{\bullet}$. M' is also called *directly reachable* from M with firing of t, denoted as $M[t\succ M'$.

A finite occurrence (of firing) sequence is $M_1[t_1\succ M_2[t_2\succ M_3 ... M_{n-1}[t_{n-1}\succ M_n$ where $M_i[t_i\succ M_{i+1}$, $1\leq i\leq n$. Marking M_1 is called *start marking* of the occurrence sequence, while M_n is called the *end marking*. The non-negative integer $n-1$ is called the number of steps in the occurrence sequence.

Definition 2.5 (Reachable)

A marking M_n is *reachable* from a marking M_1 iff there is a finite occurrence sequence whose start/end markings are M_1 / M_n correspondingly

$$M_1 [t_1 \succ M_2 [t_2 \succ M_3 ... M_{n-1} [t_{n-1} \succ M_n$$

M_n is reachable from M_1 in $n-1$ steps. The set of markings which are reachable from M_1 is denoted by $[M_1 \succ$.

2.1.1. Advantages of PNets Adoption

Many researches [29][30][31][32][33][34][39] proposed workflow modeling and analysis paradigms based on PNets, e.g., control/data flow modeling [31][32][33], workflow pattern composition [35][36][37][46], and automatic control of workflow process [38]. Aalst and ter Hofstede [39] proposed a WorkFlow net (WF-net) based on PNets to model a workflow: Transitions represent activities, places represent conditions, tokens represent cases (process instances), and directed arcs connecting transitions and places. Concluding by Aalst [40], the advantages of adopting PNets to analyze process are : (1) presenting a process with formal expression keeps the verifiability of PNets, (2) utilizing its own state-based modeling power to present process state transitions is straight forward and (3) the abundance of analysis techniques associated with PNets are available. Furthermore, **Advantage (1)** indicates that a process specification presented mathematically holds the explicitness and generality, i.e., the process can be verified by but not depends on particular tools.

Advantage (2) means that with PNets, the state transitions of the elements, task and sub-process, within workflow are expressible. In other words, PNets allows to (a) identify tasks which are enable or executing, (b) present resource competition during an execution and (c) present a cancellation of process instance by removing tokens.

Advantage (3) the available analysis techniques in control flow dimension are focused on correctness issues of control structure in a workflow. The techniques of detecting common control-flow anomalies, including deadlock, livelock (infinite loop), lack of synchronization, and dangling reference [28], are available.

Although, the three advantages reduce the difficulty of modeling and analyzing workflow application, PNets is not good enough to handle a business process presented by BPMN [34]. The expression limitations of PNets are discussed in Chapter 2. Moreover, these problems were seldom addressed in the past and were not concerned in the designs of commercial tools, e.g., Microsoft office visio [25] and BPM Virtual Modeling Tool [26].

2.1.2. Business Process Modeling Notation – BPMN

In this thesis, our process model is designed based on the core elements set specified in BPMN specification v1.2 [2], released in 2009. A business process diagram, composed of the BPMN elements, is referred to as a BPD in the following sections.

The core elements are classified into four categories, *flow objects*, *connecting objects*, *artifacts* and *swimlanes*, where

- Flow Objects: are the elements used to define the behaviour of a business process. There are three flow objects: *events*, *activities*, and *gateways*. This thesis, extended our previous work [11], presents a process model for describing the processes presented with BPMN. The term “Control node” is adopted in our previous work to present gateways. In order to keep the consistency of terminology, “gateway” is called “control node” in this thesis also.

- Connecting Objects: define the ways of connecting flow objects. There are three connecting objects: *sequence flow*, *message flow* and *association*. The execution of a BPMN-based process is controlled not only by sequence flow, the order of activities, but also by message flow, e.g., a message arriving to trigger the execution of the target flow object, as well as by the resources required to enable activities. Upon the same reason mentioned above, the term "sequence flow" is called "control flow" and artifact "association relationship" is denoted with "data flow" here.
- Artifacts: depict the information involved in a process. Within a process, what artifact is required/generated before/after an execution of activity are depicted in data flow.
- Swimlanes: The specific processes designed for a participating business role (e.g., a buyer, seller, or manufacturer) or entity (e.g., a company) can be grouped with swimlane. The process contained in a swimlane is called *private process*.

2.1.3. Problems of Modeling Processes with PNets

A workflow management system (WfMS) does not execute tasks but merely coordinates the execution of these tasks by participants or involved software systems. In a process instance, each task needs to be enabled before execution, but an enabled task does not have to execute. The execution of a task is triggered by the participants or the software systems and not by the WfMS. In the other word, a WfMS does not control the environment but reacts to events generated from the environment, e.g., instantiate a process or terminate a scheduled task, by creating certain effects, such as "a process is instantiated" or "a scheduled task is terminated". A reactive system is usually modeled using *event-condition-action rules*, stating the actions with which the system responds to events. A reactive system must respond to events in the environment with the actions specified in its rules.

Unfortunately, PNets and its higher-level extensions can model a closed active system under token-game semantics well only, but a WfMS, a reactive system, is actually open [40]. In other word, the information about the interactions between participants and their WfMS is not transformed into PNets. The omissions are summarized in Problem 1.

> Problem 1. (Interaction Omission)
>
> The interaction between a workflow management system and involved participants or systems is not captured by PNets.
>
> 1-1. The behavior of WfMS is not modeled by PNets.
>
> 1-2. An event generated from participants to enable a transition of WfMS must be fired immediately; otherwise, the system fails to respond the event.
>
> 1-3. The tasks enabled by WfMS are executed by participants or systems. But, these executions are not necessary.

When a process is modeled with a PNet, the behavior of the WfMS, on which the process executes, may not be included. Thus, the behavior simulated upon the PNet could be different from the corresponding executed at run time. The analysis results gained upon the net might be unavailable. Besides, a reactive net [41] has been proposed by extending PNets with reactive semantics; however, the indirect data presentation problem, discussed in the next two paragraphs, inherited from PNets was not addressed.

Modeling a complex business process with a PNet, holding identical tokens, could generate a large-sized PNet. During modeling, a large net could increase the difficulty of handling its complexity as well as analyzing its net structure [29][32]. For example, let a process contain many similar parts, but not identical. Using PNets, these parts must be represented by disjoint subnets of a nearly identical structure. The total PNet becomes very large. Besides, a property such as the similarities among the subnets would be very difficult to find.

All the places in a PNet are identifiable. Distinguishing the tokens based on the places cannot present data types directly, especially for an application such as workflow whose data flow is modeled with explicit channels. Comparing with Colourd Petri Nets [22], a PNet can only use more places and transitions to present data transmissions or variations. In order to indicate what and how typed data are handled in a process without complicating the net structure, there are many researches [42] using CPNets to model workflow application.

Based on our previous work [11], the artifacts involved in a process are defined to be operated by a set of legal operations, *initialize*, *read*, *update* and *destroy*. After an operation, an artifact state is transformed among the followings: *UnInitialized*, *Initialized*, *Updated* and *Read*. The correlations, existing between the operations and state transitions, can be constructed by guard and arc expressions and maintained during execution within CPNets. However, when the number of data types increases, the possible operations and their correlative state transitions are added correspondingly. Thus, constructing and maintaining the correlations with CPNets is more difficult. For example, let a process involve many different data types. Using CPNets, the correlations between the possible operations and the state transitions of all typed data need to be described in guard and arc expressions. These expressions are distributed over the CPNet. For a data type, the corresponding state transitions of its instance(s) are hard to extract. Therefore, verifying the correctness of the state transitions is difficult.

Problem 2. (High Difficulty of Maintaining Correlations)

The correlations between the artifact state transitions and legal operations within a process are not easy to be described with PNets or CPNets, because the restrictions of artifact state transitions listed in the followings are difficult to express with the two nets.

3-1. A legal operation definitely triggers an artifact state transition; even the former and latter states are identical.

3-2. Except UnInitialized state, for each state, there is one or more sequence of operations to transform the artifact from UnInitialized state to the state.

3-3. No matter which state an artifact is at, the artifact can be transformed into UnInitialized state with one operation.

In addition, when a process is modeled with BPMN, there are four different cases to introduce time conditions into the process. The four cases are: (1) inserting a timing start event to indicate the belonging process is started when a specific time condition occurs, (2) inserting a timing intermediate event into a sequential control flow to create a delay, (3) attaching a timing intermediate event to the boundary of an activity to create a deadline or time-out condition and (4) using a timing intermediate event as part of an event-based gateway. These time conditions could denote a specific or recurring time. Unfortunately, PNets and CPNets can model a process without taking time condition into account only. In other word, the information about the time conditions of a process with BPMN cannot transformed into PNets or CPNets. The omissions are summarized in Problem 3.

Problem 3. (Time Condition Omission)

The time condition(s) associated with timing start or intermediate event is not captured by PNets or CPNets.

3-1. Case (1) and (2) indicate that their implementations start to execute/continue when their corresponding time conditions are satisfied.

3-2. Case (3) indicates that the activity involved a timing intermediate event needs to accomplish before the time condition denoted.

3-3. Case (4) indicates that the outflow of an event-based exclusive gateway, started with a timing intermediate event, is selected to run when the event occurs first.

The activities in a process modeled with BPMN are either atomic or compound. A compound activity, is known as a *sub-process*, can be broken down into a finer level. BPMN can be used to create a process with different degrees of details. However, the Petri Nets do not provide a function of structuring a complex net by replacing an element (place or transition) at a higher-level of abstraction with a lower-level, more detailed, subnet.

Problem 4. (Un-introduce element refinement mechanism)

The PNets and CPNets weakly support representing a process with BPMN constructed with a sub-process which is associated with a lower-level net, especially for *Standard* and *MultiInstance* loop sub-processes.

2.2. Coloured Petri Nets — CPNets

A CPNet [22][23] allows modeling the identity of individual tokens by attaching values (or colour) to tokens. The data value may be of a primitive or a complex type as a record in PASCAL. The coloured tokens enable the modeling complicated of objects in the net. The number of the coloured token operated by a transition is assignable. The value of token(s) and its numbers in a place may be changed upon the design when one of its preceding and succeeding transition(s) is fired, i.e., the transition is defined with more elaborate operation.

This section applies the CPNet (named as *net* and shown in Figure 2.2), designed with four places, two transitions and four tokens, to explain how a CPNet works. The value of a *U*-type token located in place p_0 is x and the value of *I*-type tokens located in place p_1 is 0 and 1 and in place p_3 is 1. The value fields of *U* and *I* data type are $\{x,y\}$ and $\{0,1,2\}$, respectively.

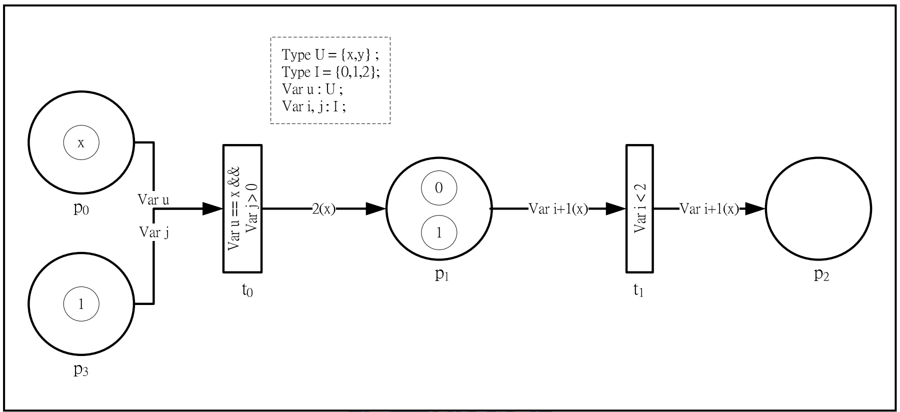


Figure 2.2 An example of CPNet.

Definition 2.6 (Coloured Petri Nets – CPNets)

A Coloured Petri Net is a 9-tuple $CNet = (P, T, F, \Sigma, C, V, A, G, m_0)$ where

1. P is a finite set of places,
2. T is a finite set of transitions,
3. F is a finite set of directed arcs, $F \subseteq (P \cup T) \times (P \cup T)$, satisfying

   $F \cap (P \times P) = F \cap (T \times T) = \phi$,
4. $\sum$ is a finite set of non-empty types, called *color sets*,
5. C is a color function, $C : P \to 2^{\Sigma}$, defined from P into the power set of $\sum$,
6. V is a finite set of variables declared by the types in $\sum$,
7. A is an arc expression function, $A : F \to \exp$ such that

$\forall f \in F : \left[Var(A(f)) \subseteq \Sigma \wedge Type(Var(A(f))) \subseteq C(p(f)) \right]$ [1].

8. G is a guard function, $G : T \rightarrow \exp$ such that $\forall t \in T$

   (1) $Type(G(t)) = Boolean$ [2],

   (2) $Var(G(t)) \subseteq \Sigma$,

   (3) $Var(G(t)) = \bigcup_{p \in {}^\bullet t} Var(A(p,t))$ and

   (4) $\forall p_1, p_2 \in {}^\bullet t, Var(A(p_1,t)) \cap Var(A(p_2,t)) = \phi$.

9. m_0 is an initialization function, $m_0 : P \rightarrow \exp$, i.e., $\forall p \in P$, $m_0(p)$ can be represented with a multi-set[3] over VE_p, defined below. By taking a type $c \in C(p)$, a *value element* associated with p is a pair (c, val) where val is one of the colors in color set c. The set of all value elements of p is denoted by $VE_p = \{(c, val) \mid c \in C(p) \wedge val \in c\}$.

The data types associated with a place p are defined as a *place color domain*, denoted as $C(p)$. The place color domains of net are $C(p_0) = \{U\}$, $C(p_1) = \{U, I\}$, $C(p_2) = \{U, I\}$ and $C(p_3) = \{I\}$. All place color domains of a CPNet are included in Σ. The tokens defined with given types included in $C(p)$ are the tokens allowing to access p only. A transition t in a CPNet is considered as a procedure with a

[1] The place connected by arc f is denoted as $p(f)$.

[2] The data type of the value returned by evaluating an expression $\exp$ is denoted as $Type(\exp)$. The set of variables in $\exp$ is denoted by $Var(\exp)$. The set of variable types used in the expression is denoted by $Type(Var(\exp))$.

[3] A multi-set m, over a non-empty set S, is a function $m : S \rightarrow \mathbb{N}$. The integer $m(s) \in \mathbb{N}$ is the number of appearances of the element s in the multi-set m.

pre-condition, declared by a guard expression, denoted as $G(t)$. The variables associated with the expression of t are defined in its *transition variable domain*, denoted as $Var(G(t))$. In net, the transition variable domains of t_0 and t_1 are $\{u,j\}$ and $\{i\}$, respectively. In addition, each variable in $Var(G(t))$ is adopted once in one of t's input arc expressions, e.g., in net, the variable u is used in t_0's input arc expression, $A(p_0,t_0)=\text{var } u$, only. For a variable v adopted in an arc expression $A(p,t)$, $Type(v)$ needs to include in $C(p)$.

Assigning the variables of a transition t with values is called *transition binding*, defined in Definition 2.7. All bindings satisfying t's guard expression are stored in $B(t)$. The form of binding b can be represented as $b=\langle v_1=val_1,v_2=val_2,\ldots,v_n=val_n\rangle$ where v_i is assigned with value val_i, $Var(G(t))=\{v_i \mid 1\le i\le n\}$. In net, there are two bindings $b_1=\langle i=3\rangle$ and $b_2=\langle i=5\rangle$ in $B(t_1)$.

Definition 2.7 (Transition Binding)

A *binding* of a transition t is a function $b:Var(G(t))\rightarrow M$, M is defined in Definition 2.8, where $\forall v\in Var(G(t))$

1. $b(v)=1(p,(c,val))$, i.e., the value val of the c-typed token in p is assigned to variable v in $A(p,t)$ and replaces v of $G(t)$, and
2. $Type(v)=c$, i.e., the type of variable v is the same as that of the selected token.

A *token element* is a pair $(p,(c,val))$ where $p \in P$ and $(c,val) \in VE_p$, while a *binding element* is a pair (t,b) where $t \in T$ and $b \in B(t)$. The set of all token elements of a CPNet is denoted by TE while the set of all binding elements is denoted by BE. In net, the color sets associated with p_1 and p_2 are U and I while p_0 and p_3 are associated with U and I, respectively. The TE of net is composed of the token elements in the two sets,

$\{(p,(U,x)),(p,(U,y)) \mid p=(p_0 \mid p_1 \mid p_2)\}$ and

$\{(p,(I,0)),(p,(I,1)),(p,(I,2)) \mid p=(p_1 \mid p_2 \mid p_3)\}$.

The BE are (t_0,b_0), (t_1,b_1) and (t_1,b_2) where $b_0=\langle u=x, j=1\rangle$, $b_1=\langle i=0\rangle$ and $b_2=\langle i=1\rangle$.

Definition 2.8 (Marking)

A *marking* M is a multi-set over TE while a step Y is a non-empty and finite multi-set over BE. The *initial marking* M_0 is obtained by initialization function m_0:

$$\forall (p,(c,val)) \in TE : M_0(p,(c,val)) = (m_0(p))(c,val).$$

The set of all markings and steps are denoted by $\tilde{M}$ and $\tilde{Y}$, respectively.

Definition 2.9 (Step Enabled)

A step Y is enabled in a marking M, obtained by a marking function m, if and only if the following property is satisfied:

$$\forall p \in P : \sum_{(t,b)\in Y} A(p,t)\langle b\rangle \subseteq m(p)$$

Let $(t,b)\in Y$. The tokens in $A(p,t)\langle b\rangle$, a multi-set over VE_p yielded by the arc expression $A(p,t)$ upon b, are removed from p when t is fired with binding b. By taking all binding elements $(t,b)\in Y$, the tokens in the union of multi-sets generated by these binding elements are removed from the input places concurrently when Y occurs. Each binding element (t,b) in Y must be able to get the tokens specified by $A(p,t)\langle b\rangle$, without having to share these tokens with other binding elements of Y.

Let step Y be enabled in the marking M. When $(t,b)\in Y$, t is enabled in M with the binding b. If $(t_1,b_1),(t_2,b_2)\in Y$ and $(t_1\neq t_2)\wedge(b_1\neq b_2)$, (t_1,b_1) and (t_2,b_2) are enabled concurrently in marking M. If $|Y(t)|\geq 2$, i.e., $\exists i,j$ $(t,b_i),(t,b_j)\in Y$ and i may be j, t is enabled more than one time concurrently.

Definition 2.10 (Fire a Step)

When a step Y is enabled in a marking M_1, generated by marking function m_1, marking function m_2 generating the next marking M_2 from M_1 can be defined as:

$$\forall p\in P: m_2(p)=\left(m_1(p)-\sum_{(t,b)\in Y} A(p,t)\langle b\rangle\right)+\sum_{(t,b)\in Y} A(t,p)\langle b\rangle$$

Multi-set $\sum_{(t,b)\in Y} A(p,t)\langle b\rangle$ represents the tokens removed from p, while $\sum_{(t,b)\in Y} A(t,p)\langle b\rangle$ denotes the tokens added to p. M_2 is directly reachable from M_1 by the occurrence of the step Y, denoted as $M_1[Y>M_2$.

The initial marking M_0, generated by m_0, of *net* is $1'(p_0,(U,x))+1'(p_1,(I,0))+1'(p_1,(I,1))+1'(p_3,(I,1))$. Let two sequential steps Y_1 and Y_2 be $\{(t_0,b_0)\}$ and $\{(t_1,b_1),(t_1,b_2)\}$. Before executing Y_1, the values of the tokens, $1'(p_0,(U,x))$ and $1'(p_3,(I,1))$, are assigned to variable u and j upon $b_0=\langle u=x,j=1\rangle$ for evaluation, i.e., u is assigned with x of the token in place p_0, while j is assigned with value 1 of the token in place p_3. In this case, the evaluation result is true, hence Y_1 is enabled in M_0 and it may be fired. When Y_1 is fired, one U-type token with value x and one I-type token with value 1 are removed from p_0 and p_3, respectively, and two U-type tokens with value x are added into p_1. The result is shown in Figure 2.3.

In Y_2, transition t_1 is enabled twice concurrently by binding $b_1=\langle i=0\rangle$ and $b_2=\langle i=1\rangle$, i.e., the two binding elements in Y_2 are able to remove the corresponding tokens, expressed as $1'(p_1,(U,x))+1'(p_1,(I,0))$ and $1'(p_1,(U,x))+1'(p_1,(I,1))$ respectively, from p_1 at the same time.

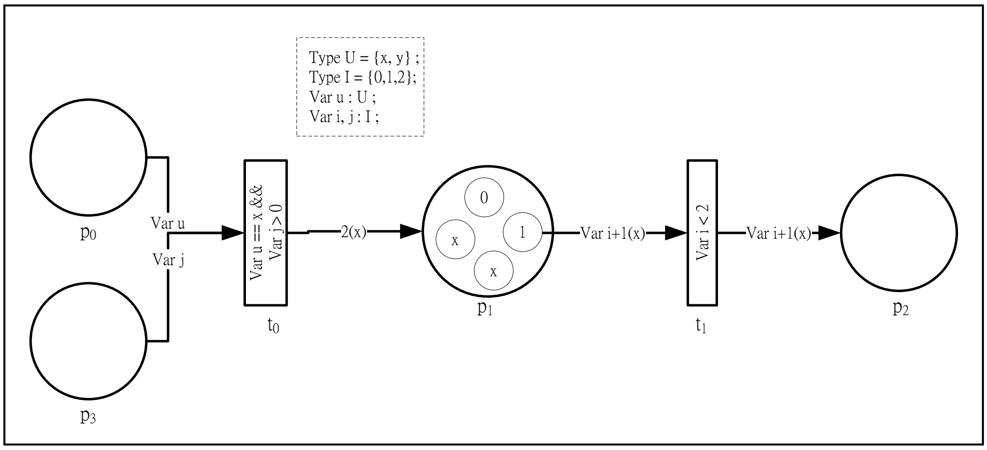


Figure 2.3 The result net of firing step Y_1.

The definition of occurrence sequence of CPNets, omitted here, is similar to that of PNets, given in Definition 2.5.

2.3. Time Coloured Petri Nets — TCPNets

CPNets with timing constraints can be classified according to the ways of specifying timing constraints, a timing interval [16][17][23] or a specific time [18], or the elements of the net, place [19] , transition [16][18] and arc [15][20], these constraints are associated with. When timing constraints are associated with a transition, the constraint can be interpreted as (1) a *delay time* [18] [23], i.e., when the transition is fired, its input tokens are removed, but the output tokens is created until the delay time associated with the transition has elapsed, (2) a *holding duration* [21], i.e., when the transition is fired, its input and output tokens are removed and added concurrently, but the succeeding transition is enabled when the token created time within the holding duration denoted and (3) an *firing interval* [16][23], i.e., the transition can be fired in its firing interval only. For such transition, the mechanism of removing and adding tokens is the same as that of a transition associated with a delay time.

A common approach [23] is to associate a *time stamp*, denoted as $@r$, $r \in \mathbb{R}$, with token, and attach a *restricted firing interval*, denoted as $[min,max]$, $min,max \in \mathbb{R}$, with transition. The transition output arc(s) can associate with a *time requirement* Δt to denote how many time units an execution of the transition takes.

When a token is associated with a time stamp, the token is *timed*. If the time stamp is $@r$, the token is available to consume after r, i.e., r is the earliest time at which the token can be used. Otherwise, the token is *untimed* and always ready to be used. For a *timed transition* t, there is a restricted firing interval $[min,max]$ associated with t which is a pair of real numbers referred to the *minimum* and *maximum firing time*, respectively, i.e., t can be fired between min and max only. In addition, an execution of t takes Δt time units which is equal to or more than 0. Δt is specified in t's output arc expression(s). An untimed transition, defined without restricted firing interval, can be fired when it is enabled. The firing mechanism of untimed transition is the same as that defined in CPNets.

In a TCPNet, timed CPNet, a *global clock* is introduced. Let an activity, associated with a restricted firing interval $[min,max]$, be presented with a transition t in the net and t be fired at τ, $min \le \tau \le max$. An execution of t takes Δt time units. The value of the time stamp(s) associated with the token(s), which will be removed from t's input place(s) when t is fired, needs to be less than τ. When t is fired, t creates a time stamp $\tau + \Delta t$ for its output token(s).

Definition 2.11 (Timed Coloured Petri Nets)

A Timed Coloured Petri Net is a 5-tuple $TNet = (CNet, I_{INT}, I_R, R, r_0)$ where

1. $CNet = (P,T,F,\Sigma,V,C,G,A,m_0)$ is a CPNet where
   (1) $\Sigma = \Sigma_U \cup \Sigma_T$, i.e., the colour sets (types) in Σ can be divided into two disjoint sets, Σ_U and Σ_T. The elements in Σ_U are untimed and the elements in Σ_T are timed, i.e., a token typed with $c \in \Sigma_T$ is associated with a time stamp,
   (2) $\forall f \in F$, the variables $Var(A(f))$ used in arc f are timed/untimed over the timed/untimed subset of $C(p(f))$ and
   (3) $\forall p \in P$, $m_0(p)$ generates a timed/untimed multi-set over the timed/untimed subset of $C(p)$. The details are given in Definition 2.12.
2. I_{INT} is an interval function $I_{INT}: T \rightarrow INT$ where $INT = \{[x,y] \in \mathbb{R} \times \mathbb{R} \mid x \le y\}$.

For a transition t, $t \in T$, the function assigns a firing interval $[min,max]$.

3. I_R is a time function $I_R : F \rightarrow R$ where $R = \{\Delta t \in \mathbb{R} \mid \Delta t \geq 0\}$. For an arc (t,p), $(t,p) \in F$, the function assigns the time units consumed by executing t on (t,p).
4. R, $R \subset \mathbb{R}$, is a set of time values, called *time stamps*.
5. r_0, $r_0 \in R$, is the start time.

The definitions of the set of transition bindings $B(t)$, token elements TE, binding elements BE and step Y are the same as those of CPNets.

This section applies the TCPNet net (shown in Figure 2.4), designed with four timed tokens and two timed transitions, to explain how a TCPNet works. We declare that R includes time stamps 100, 200 and 220. There are four tokens typed with the colour sets in Σ, $\Sigma = \Sigma_T$. The U-typed token, which is assigned with value x and located in place p_0, is available after time 100. The three I-typed tokens, which are assigned with value 0, 1, 1 and located in place p_1, p_1, p_3, are available at 100, respectively. Transition t_0 and t_1 are associated with restricted firing intervals $[180,220]$ and $[200,250]$, respectively. An execution of t_0/t_1 takes 20/30 time units.

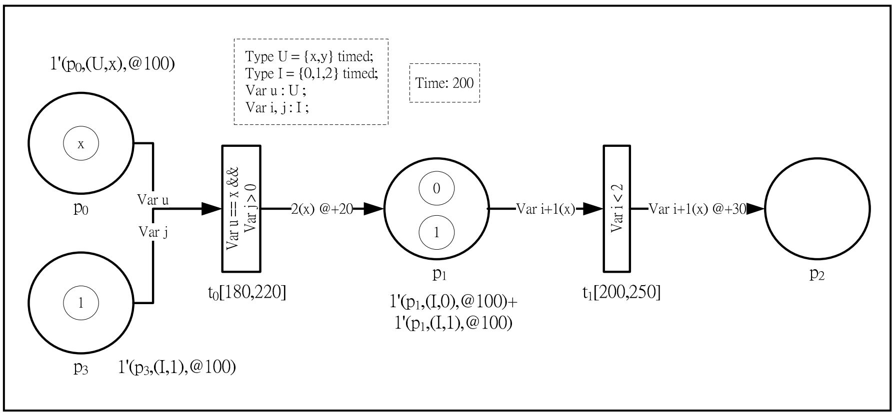


Figure 2.4 Introducing time constraints into the CPNet shown in Figure 2.1.

Definition 2.12 (Timed Multi-set)

A timed multi-set tm, over VE_p of a place p in $CNet$, is a function $tm: VE_p \times R \rightarrow \mathbb{N}$, such that

1. $tm_{(c,val)} = \sum_{r \in R} tm((c,val),r)$, an non-negative integer, denotes the number of c-typed tokens associated with val in p,
2. the time stamps associated with these c-typed tokens are listed in

   $tm[(c,val)] = [r_1, r_2, \ldots, r_i, \ldots, r_{tm_{(c,val)}}]$

   where the time value r_i for $tm((c,val),r_i) \neq 0$, $1 \leq i \leq tm_{(c,val)}$, are listed. r_i appears $tm((c,val),r_i)$ times in the list and $tm[(c,val)]$ is sorted, i.e., $r_i \leq r_{i+1}$, $1 \leq i \leq tm_{(c,val)}$.

A formal presentation of tm of p is $\sum_{(c,val) \in VE_p} tm_{(c,val)}{'}(c,val)@tm[(c,val)]$.

In net, formal presentations of the tokens located in place p_0, p_1, p_3 are $1'(U,x)@[100]$, $1'(I,0)@[100]+1'(I,1)@[100]$ and $1'(I,1)@[100]$, respectively.

Definition 2.13 (Timed Marking)

Given a Timed CPNet $TNet = (CNet, I_{INT}, I_R, R, r_0)$, a *timed marking* (state) of $TNet$ can be denoted by a pair (M,r), the untimed marking M is a multi-set over TE of $CNet$ and generated by marking function m at time r such that

$$\forall (p,(c,val)) \in TE: M(p,(c,val))_r = (m(p)_r)(c,val).$$

The *initial timed marking* can be denoted by a pair (M_0, r_0). The sets of all untimed and timed markings are denoted by $\widetilde{M}_U$ and $\widetilde{M}_T$, respectively.

Upon Definition 2.13, the initial timed marking (M_0, r_0) of net is

$$\left(1'\left(p_0,(U,x)\right)+1'\left(p_1,(I,0)\right)+1'\left(p_1,(I,1)\right)+1'\left(p_3,(I,1)\right),@100\right)$$ where

$M_0 = 1'\left(p_0,(U,x)\right)+1'\left(p_1,(I,0)\right)+1'\left(p_1,(I,1)\right)+1'\left(p_3,(I,1)\right)$ and $r_0 = 100$.

Definition 2.14 (Step Enabled)

Given a Timed CPNet $TNet = (CNet, I_{INT}, I_R, R, r_0)$, a step Y of $TNet$ is enabled in a timed marking (M_1, r_1) at time r_2 if and only if the following properties are satisfied:

(1) $\forall p \in P: \sum_{(t,b)\in Y} A(p,t)\langle b\rangle_{r_2} \subseteq m_1(p)$,

(2) $r_1 \le r_2$,

(3) r_2 is the smallest value of R which satisfies (1) and (2).

Let step Y of $TNet$ be enabled in (M_1, r_1) at the smallest time r_2 in R, $r_1 \le r_2$. For each binding element $(t,b) \in Y$, the tokens in $A(p,t)\langle b\rangle$, a multi-set over VE_p yielded by the arc expression $A(p,t)$ upon b at time r_2, are associated with time stamps whose values are equal to or smaller than r_1.

The set of time stamps of net, marked with (M_0, r_0) where $r_0 = 100$, is $R = \{100, 200, 220\}$. Let two sequential steps Y_1 and Y_2 of net be $\{(t_0, b_0)\}$ and $\{(t_1, b_1), (t_1, b_2)\}$. The two steps are enabled at r_1 and r_2, respectively. The restricted firing intervals of transition t_0 and t_1 are $[180, 220]$ and $[200, 250]$. In Section 0, the two sequential steps can be fired sequentially without concerning time constrains. Here, we concern the firing intervals of transition t_0 and t_1.

For the case of Y_1, Y_1 is enabled when $r_1 = 200$ only, because '200' is the only time stamp in R between firing boundary 180 and 220 of t_0. If Y_1 is fired at τ_1, $200 \le \tau_1 \le 220$, one U-type token with value x and one I-type token with value 1 are removed from p_0 and p_3, respectively, and two U-type tokens with value x are added into p_1. A time stamp $@\tau_1 + 20$ is created for the two added tokens. The timed marking of the result net, shown in Figure 2.5, is $1'(p_1,(I,0))@100 + 1'(p_1,(I,1))@100 + 2'(p_1,(U,x))@\tau_1 + 20$. After firing Y_1, Y_2 can be enabled at $r_2 = 220$, if $\tau_1 + 20 \le 220$. For the case of Y_2, Y_2 can be enabled when $\tau_1 = 200$ only. If Y_2 is fired at $\tau_2 = 220$, the two binding elements (t_1, b_1) and (t_1, b_2) in Y_2 are able to remove the corresponding tokens, expressed as $1'(p_1,(U,x))@220 + 1'(p_1,(I,0))@100$ and $1'(p_1,(U,x))@220 + 1'(p_1,(I,1))@100$ respectively, from p_1 at the same time. A time stamp $@220 + 30$ is created for the four tokens generated by t_1 and added into p_2.

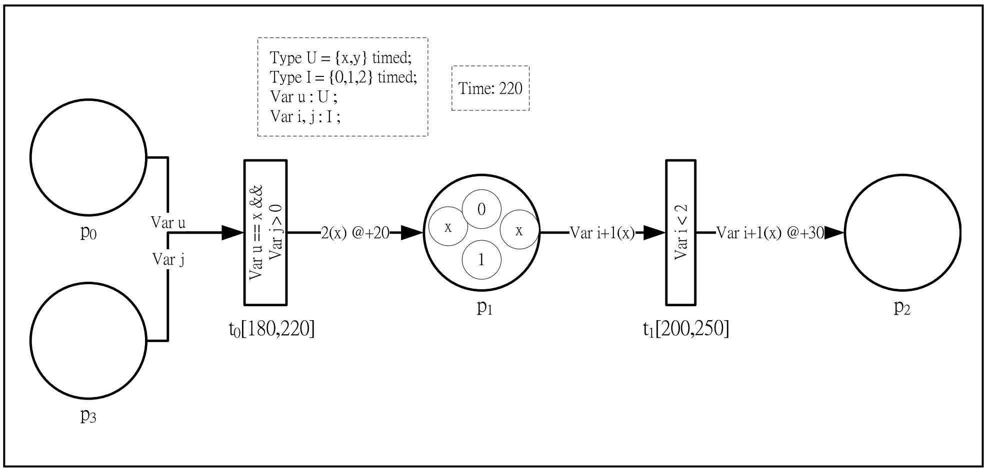


Figure 2.5 The result net of firing step Y_1.

Definition 2.15 (Fire a Step)

When a step Y is enabled in a timed marking (M_1,r_1) at time r_2, generated by marking function m_1, marking function m_2 generating the next marking (M_2,r_2) from (M_1,r_1) can be defined as:

$$\forall p \in P: m_2(p) = \left(m_1(p) - \sum_{(t,b)\in Y} A(p,t)\langle b \rangle_{r_2} \right) + \sum_{(t,b)\in Y} A(t,p)\langle b \rangle_{r_2}$$

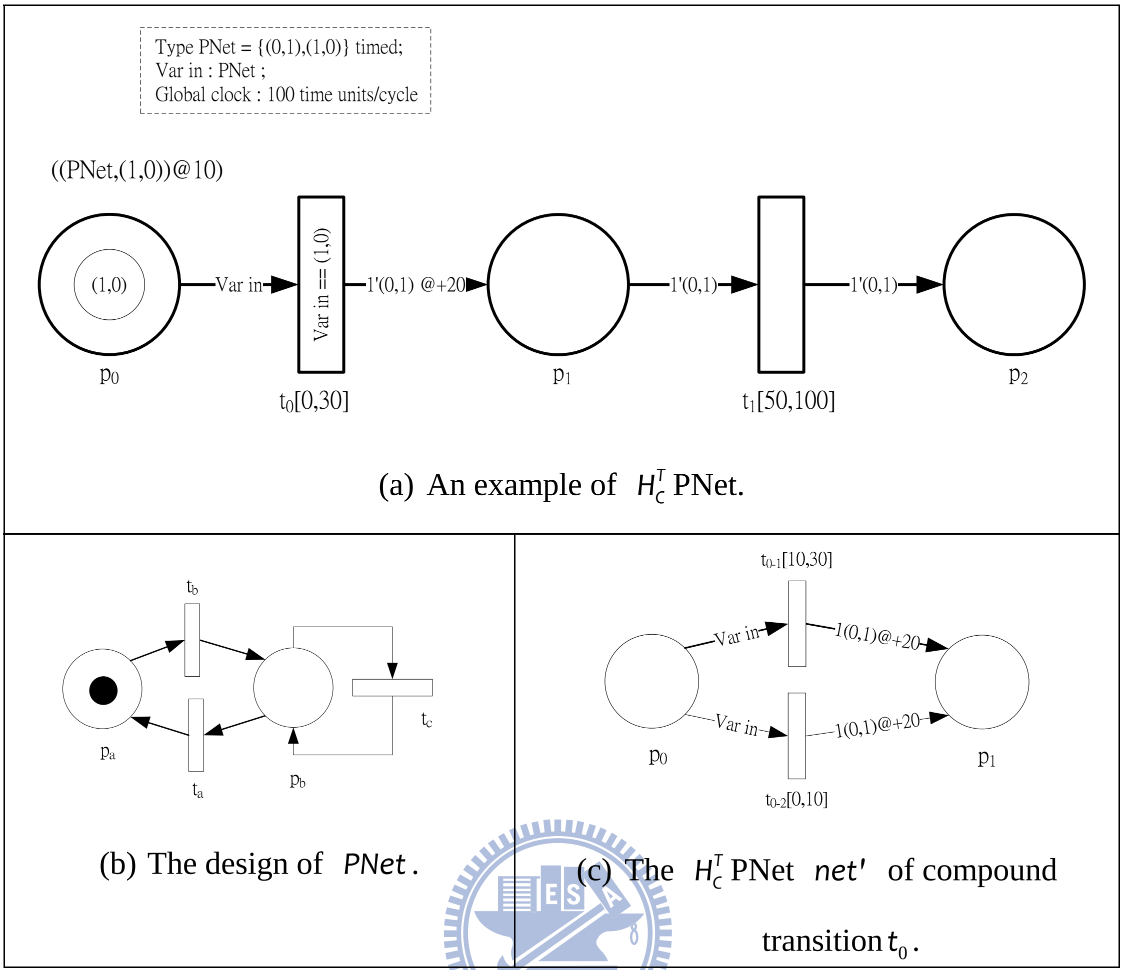
Multi-set $\sum_{(t,b)\in Y} A(p,t)\langle b \rangle_{r_2}$ represents the tokens removed from p, while $\sum_{(t,b)\in Y} A(t,p)\langle b \rangle_{r_2}$ denotes the tokens added to p. (M_2,r_2) is directly reachable from (M_1,r_1) by the occurrence of the step Y, denoted as $(M_1,r_1)[Y,r_2 > (M_2,r_2)$.

2.4. Timed CPNets with Hierarchy – H_C^T PNets

A H_C^T PNet defined in Definition 2.16 is a Timed CPNet with hierarchy, which is defined as the followings:

1. Hierarchical Transition: A transition t in a H_C^T PNet can denote a collapsed sub-process whose expansion is another H_C^T PNet. The pre-condition associated with t has to be met before the execution of t's corresponding net.

2. Hierarchical Token: Each token in a H_C^T PNet is typed with a Petri net $PNet$, called *PNet type*, accompanied an initiation marking M_0. The set of markings $[M_0 \succ$, reachable from M_0, is the color set of $PNet$ type.



(a) An example of H_C^T PNet.

(b) The design of $PNet$.

(c) The H_C^T PNet net' of compound transition t_0.

Figure 2.6 An example of H_C^T PNet.

This section applies the H_C^T PNet net, shown in Figure 2.6 (a), designed with three places and two transitions, to explain how a H_C^T PNet works. Let the initial marking of net be $1'\left(p_0,\left(PNet,(1,0)\right)\right)@[10]$ and transition t_0 be compound. A $PNet$-type token is putted in place p_0 of net at time 10. The token is marked with $(1,0)$ while the place array of $PNet$ is (p_a,p_b). The compound transition t_0 can be expanded to net', shown in Figure 2.6 (c).

Definition 2.16 (Hierarchical Timed Petri Nets – H_C^T PNets)
A Hierarchical Timed Petri Net is a 5-tuple

$HNet = (TNet, TrSet, TkSet, TrFun, TkFun)$ where

1. $TNet = (CNet, I_{INT}, I_R, R, r_0)$ is a Timed CPNet, where the set of transitions T in $CNet = (P, T, F, \Sigma, C, V, A, G, m_0)$ can be divided into two disjoint sets, T_A and T_C. The transitions in T_A are atomic and the transitions in T_C are compound.
2. $TrSet$ is a finite set of H_C^T PNets each of which represents the expansion of a compound transition in T_C.
3. $TkSet$ is a finite set of PNets each of which represents the design of a data type in Σ.
4. $TrFun$ is a compound transition mapping function, $TrFun: T_C \rightarrow TrSet$, defined from T_C to $TrSet$, $TNet \notin TrSet$. The number of nets in set $TrSet$ is equal to the number of compound transitions in set T_C, i.e., $|TrSet| = |T_C|$ and $|T_C| \geq 0$. Each compound transition in T_C is mapped into one of the H_C^T PNets in $TrSet$. Function $TrFun$ is 1-1 and onto.
5. $TkFun$ is a type mapping function, $TkFun: \Sigma \rightarrow TkSet$, defined from Σ into $TkSet$. The number of nets in set $TkSet$ is equal to the number of types in Σ, i.e., $|TkSet| = |\Sigma|$ and $|\Sigma| \geq 0$. Each type (color set) in Σ is mapped into one of the PNets in $TkSet$. Function $TkFun$ is 1-1 and onto.

Definition 2.17 (Weakly Connected Net)

A net, PNet or its extension, is called *weakly connected* if and only if replacing all of its directed arcs with undirected ones produces a connected net, i.e., there is a path between any pair of distinct nodes in the net at least.

Definition 2.18 (H_C^T PNet of Compound Transition)

Given two *weakly connected* H_C^T PNets, $HNet$ and $HNet'$, $HNet \neq HNet'$, a compound transition t of $HNet$ is associated with $HNet'$, $TrFun(t) = HNet'$, if and only if the following conditions hold. Let $CNet'$ of $HNet'$ be composed of $(P',T',F',\Sigma',V',C',G',A',m_0')$.

1. The input and output places of t are transferred into P', $(In(t) \cup Out(t)) \subset P'$, i.e., $HNet'$ is started from the places in $In(t)$ and terminated at the places in $Out(t)$. There is a path between any pair of start and terminated nodes at least,
2. $|T'| > 1$, the number of transitions in T' is more than 1,
3. $\bigcup_{p \in In(t) \cup Out(t)} C(p) \subseteq \Sigma'$, the types (color sets) associated with the places in $In(t) \cup Out(t)$ are included in Σ' and
4. $\forall p \in (In(t) \cup Out(t))$, $C'(p) = C(p)$, i.e., the types associated with p in $HNet$ are the same as that in $HNet'$.

Definition 2.19 (PNet of Color Set)

Given a H_C^T PNet $HNet$ and a weekly connected Petri Net $PNet = (P,T,F,m_0)$, a type (color set) tp involved in $HNet$ is designed with $PNet$, i.e., $TkFun(tp) = PNet$, if and only if the following conditions hold:

1. $|P| \geq 1$, i.e., there is one or more place in P,

2. $\forall t \in T$, $|{}^{\bullet}t| = |t^{\bullet}| = 1$, i.e., t has exact one input and output places,

3. The initial marking function $m_0 : P \rightarrow \{0,1\}$ and $\sum_{p \in P} m_0(p) = 1$, i.e., an initial marking M_0 of $PNet$, generated by function m_0, includes one token only. From M_0, all reachable markings include one token also, $\forall M_i \in [M_0 \succ : \sum_{p \in P} m_i(p) = 1,\ 0 < i \leq n$ and $n = |[M_0 \succ|$

The number of colors in color set tp is less than or equal to the number of places in $PNet$. These colors are presented with the states in $[M_0 \succ$.

For simplicity, and without losing generality, we assume that each H_C^T PNet has two levels in its hierarchy only. When a H_C^T PNet is designed with more than two levels, the compound transitions located in higher levels, 2 or more than 2, can be recursively replaced by its finer nets. In addition, any H_C^T PNet, restricted to start and end with places, is weakly connected, i.e., there is a path between any pair of distinct nodes (places and transitions) in the net at least.

Chapter 3. Business Process Modeling

In general, a business process is implemented with one or more *private processes* (also called "process" in this thesis for short) for a business purpose. Each process is designed for a distinct business role (e.g., a buyer, seller, or manufacturer) or entity (e.g., a rule checking machine or banking system) involved. The participants acting the appointed roles cooperate according to the processes assigned to produce a product or service for a particular customer or market. Message sending is the only way to create a communication between processes. We define *messaging* as the (usually asynchronous) sending of a data item from a business role(s)/entity to other role/entity(ies). A *message flow* is used to present the transmission of messages. A business process specification, in Definition 3.1, defines the interactions between processes with message flow while the details of these processes are specified in their own specifications. The core modeling elements in BPMN are adopted and shown in Figure 3.1.

Definition 3.1. (Business Process Specification)

A business process specification is a 7-tuple $BP = (PP, A, M, MF, \widetilde{MF}, PF, \tilde{P})$, where

1. PP is a set of private processes, as defined in Definition 3.2,
2. A denotes the set of artifacts used in BP,
3. M denotes the set of messages used in BP,
4. $MF \subseteq (PP \times PP)$ is a set of directed edges, called *message flow*, indicating the sender-receiver relations,

5. $\widetilde{MF}: MF \to M$ is a message function that maps each message flow into one of the messages in M, 6. PF defines the set of resources that perform or are responsible for BP 7. $\tilde{P}: PP \to PF$ is a resource (onto) function that maps each process into one of the resources in PF.

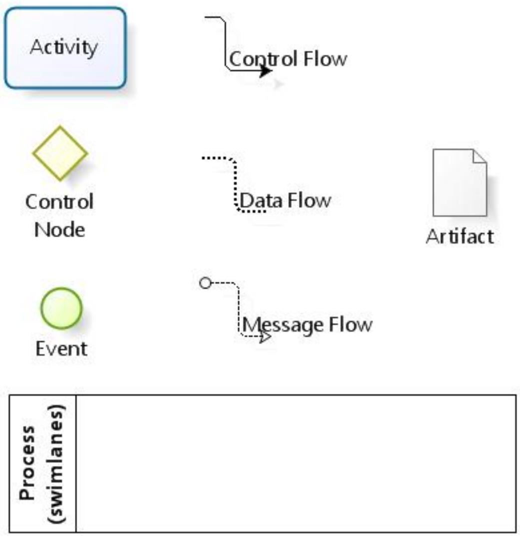


Figure 3.1 The core modeling elements in BPMN.

A business process for resolving problem through e-mail votes is applied in this thesis for demonstrating the usage of our formal model. The example is illustrated from broad to narrow.

There are three roles, working group, manager and voter, responsible for the voting business process, BP_{vote}. The assignments of the three roles, process $P_{workingG}$, $P_{manager}$ and P_{voter}, are described within their own swimlanes. The control and data flows of the three processes are introduced in Section 3.2.4 and 3.3.4, respectively. The participants acting working group, manager or voter execute $P_{workingG}$, $P_{manager}$ or P_{voter} to solve an intended problem. In the beginning of BP_{vote} execution, message "issue list" is sent from a working group to its manager. And then, the messages in set M_{vote} are transmitted between the manager and voters as the message flow shown in the business process diagram, displayed in Figure 3.2,

$$M_{vote} = \left\{ \begin{matrix} IssueAnnouncement, Vote, DeadlineWarning, \\ VoteResults, ChangeMessage \end{matrix} \right\}.$$

These message flows can be presented formally as $\widetilde{MF}\left(P_{manager}, P_{voter}\right) = IssueAnnouncement$.

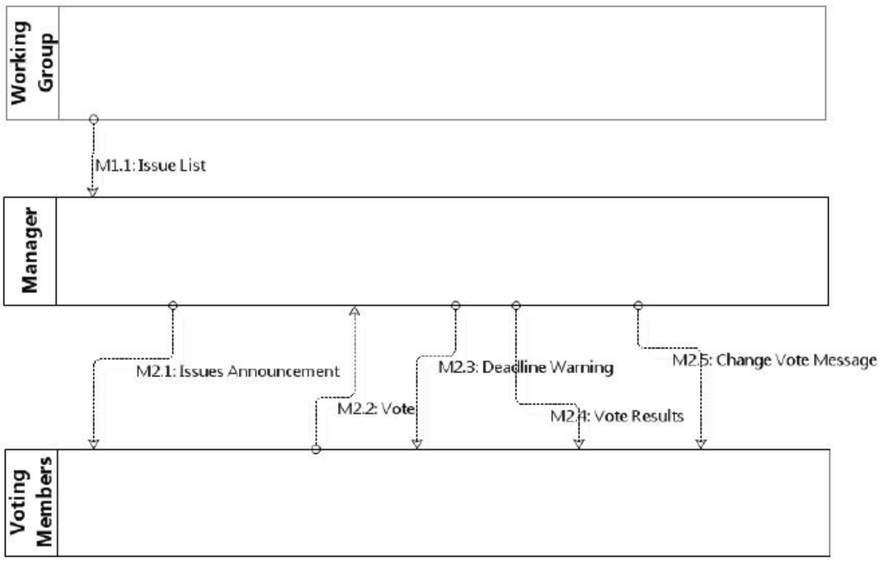


Figure 3.2 E-mail voting process

3.1. Private Process Specification

Within business process BP, a process P, associated with performer $\tilde{P}(P)$, consists of a network of actions designed to achieve part of work in BP. The specification of process P contains a *control flow* and *data flow*.

A control flow defines a set of connected (parallel and/or sequential) actions and indicates the start and end event(s) of the process. In addition, the *intermediate event*s occurring between the start and the end are described for the execution flow of process, not for its start or end. For example, when a process instance catches a time event, it can switch the execution from normal flow to some handling process.

The control flow construction mechanism proposed in this thesis contains two parts: *basic* and *supplement*. The basic construction mechanism, defined in Definition 3.3, is used to build an action network without including an activity involving event(s). Otherwise, the supplement mechanism, defined in Definition 3.12, is adopted.

A process is specified with an explicit data flow in the thesis. A data, called *artifact*, is passed from one action to another via explicit channels which are distinct from the control arc between these actions. Each action takes a subset of the process input or the output of its previous action(s) connected by the data flow and transforms them into data for next action(s) or as process outputs. The details are described in Section 3.3.

Here, we give a formal definition of private process in Definition 3.2.

Definition 3.2. (Private Process Specification)

For a given business process BP, a process P belonging to BP is specified with a tuple $P=(ControlFlow,DataFlow)$, where

1. $ControlFlow$ represents a control flow specification of process P,
2. $DataFlow$ represents a data flow specification of process P.

3.2. Control Flow Specification

Definition 3.3. (Control Flow Specification)

Given a process P, the control flow associated with P is specified with a 6-tuple $ControlFlow(P)=(G,\tilde{V},A,M,I,O)$, where

1. $G=(V,CF)$ represents the control flow of P with a directed graph, where V is a set of vertices of which each represents an action and $CF \subset V \times V$

is a set of directed edges indicating the precedence relation between two actions,

2. $\widetilde{V}: V \rightarrow T$ is type function that maps each action into one of the action types in T, where $T = \{Event, Activity, ControlNode\}$,

3. A is a set of artifacts used in P and $A \subseteq A_{BP}$,

4. $I = IA \cup IM$ is a set of process inputs, where IA, $IA \subseteq A$, denotes the set of artifact inputs and IM denotes the set of messages (sent from other processes in BP) can be caught at P,

5. $O = OA \cup OM$ is a set of process output, where OA, $OA \subseteq A$, denotes the set of artifact outputs and OM denotes the set of messages threw out from P,

6. M is a set of messages used in P, $M \subseteq M_{BP}$ and $M = IM \cup OM$.

Definition 3.4. (Predecessors and Successors).

1. $V_v^{IsPredecessor} = \{u \in V \mid (u,v) \in CF\}$

   $V_v^{\overline{IsPredecessor}} = \{t \in V \mid t \in V_v^{IsPredecessor} \vee (\exists u \in V_v^{IsPredecessor} : t \in V_u^{\overline{IsPredecessor}})\}$

2. $V_v^{IsSuccessor} = \{u \in V \mid (v,u) \in CF\}$

3. $V_v^{\overline{IsSuccessor}} = \{t \in V \mid t \in V_v^{IsSuccessor} \vee (\exists u \in V_v^{IsSuccessor} : t \in V_u^{\overline{IsSuccessor}})\}$

$V_v^{IsPredecessor}$ comprises the set of vertices which are the source of an edge with destination vertex $v \in V$. Each element u in $V_v^{IsPredecessor}$ is called a *direct predecessor* of the vertex and is denoted by $u \rightarrow v$. $V_v^{\overline{IsPredecessor}}$ denotes the transitive closure of $V_v^{IsPredecessor}$. $\forall u \in \overline{V_v^{IsPredecessor}}$, v is reachable from u. Each element u in $\overline{V_v^{IsPredecessor}}$ is called a *predecessor* of v and is denoted by $u \twoheadrightarrow v$. $V_v^{IsSuccessor}$ and its transitive closure $\overline{V_v^{IsSuccessor}}$ are defined similarly.

3.2.1 Events

In a process, an event, defined in Definition 3.5, is an action that is handled by an activity inside the process. An event affects the execution of a process; a process changes its flow in response to events. Based on the time the events affect a process, the events can be classified into three categories: *start*, *intermediate*, and *end*, defined in Definition 3.6.

Definition 3.5. (Event)

Given a process P where $G=(V,CF)$, each event in set $E=\{v \in V \mid \tilde{V}(v)=Event\}$ can be described with the attributes listed followings:

1. EC_v attribute represents the category of v, which is defined by $\tilde{E}: E \rightarrow C$, a classification function to map each event in E into one category in C, where $C=\{Start, End, Intermediate\}$.

2. ET_v attribute represents the type of v, which is defined by $\tilde{E}: E \rightarrow T$, a type function to map each event in E into one type in T, where $T=\{None, Message, Time\}$.

3. $Timer_v$ is an attribute to represent the timer set on v. The default value of $Timer_v$ is $None$.

4. $InMessage_v$, $InMessage_v \in IM_P$, is an attribute to represent the message

expected to receive on v. The default value of $InMessage_v$ is $None$.

5. $OutMessage_v$, $OutMessage_v \in OM_P$, is an attribute to represent the message dispatched on v. The default value of $OutMessage_v$ is $None$.

- Start Event

An event is a *start event* if only if when the trigger for the event occurs, a process belonged is instantiated and a token is generated with identification for that instance.

- Intermediate Event

An event is an *intermediate event* if only if the event happens between the start and end of a process. The event affects the flow of process, but does not start or terminate the process. It can be used to show where messages are expected/sent or where action delays are defined.

- End Event

An event is an *end event* if only if the event ends the process by consuming the token generated from a start event.

Definition 3.6. (Categories of Event)

Given a process P defined by control flow $G=(V,CF)$, the events belonging to P are in $E=\{v \in V \mid \tilde{V}(v)=Event\}$. E can be divided into three disjoint sets, $StartSet$, $EndSet$, and $InterSet$, such that

- $StartSet=\{v \in E \mid EC_v = Start \wedge (InDegree(v)=0 \wedge OutDegree(v)>0)\}$[4],
- $EndSet=\{v \in E \mid EC_v = End \wedge (InDegree(v)>0 \wedge OutDegree(v)=0)\}$,

[4] Function $InDegree$ and $OutDegree$ are used to denote the number of incoming and outgoing control flows of action.

- $InterSet = \{v \in E \mid EC_v = Intermediate \wedge (InDegree(v) = 1 \wedge OutDegree(v) = 1)\}$

The number of events in $StartSet$ and $EndSet$ is more than 0.

There are many cases which could be considered as an event, e.g., the start of an activity, the state change of a document or the end of a process. To simplify the discussion, we restrict the use of events to include only those message or timing events that affect the sequence or timing of activities of a process. The event types concerned in our model are: *none*, *message* and *time*. How these events are executed in a process is described in the followings:

- None event

When a process execution reaches an event node which is denoted with *none*, the event occurs immediately. A formal definition of *none event* is given in Definition 3.7. In general, this kind of event is a start or end event, because an intermediate event denoted with none is omissible. Thus, if a process modelled with *none start* or *intermediate event*, the process can be instantiated right away or terminated immediately when reaching the end. The notations for none event in BPMN are adopted and shown in Figure 3.3.

Definition 3.7. (None Event)

Given a process P, a $StartEvent$ of P instantiates the process without waiting for a trigger if only if the following condition holds:

- $\exists StartEvent \in P.StartSet : StartEvent.Timer = None \wedge StartEvent.InMessage = None \wedge StartEvent.OutMessage = None$.

An $EndEvent$ of P terminates an instance when reaching the end if only if the following condition holds:

■ $\exists EndEvent \in P.EndSet : EndEvent.Timer = None \wedge EndEvent.InMessage = None \wedge EndEvent.OutMessage = None$.

Figure 3.3 The notations for none events.

■ Message Event

When a process execution reaches an event node which is denoted with *message*, the process continues upon when the message is received or submitted. If the event node is a *message start event*, the process starts to wait for an inserting message. When the message trigger for the event occurs, a new process instance is generated. If the event node is a *message intermediate event*, there are two possible scenarios. Firstly, the process is blocked till an expected message is received. Secondly, a described message is dispatched. The notations of a message intermediate event associated with *receiver* and *dispatcher* are presented in Figure 3.4(a) and (b), respectively. If the event node is a *message end event*, the process dispatches a message at the end of process. A formal definition of *message event* is given in Definition 3.8. Notations for the message events in BPMN are adopted and shown in Figure 3.4.

Definition 3.8. (Message Event)

Given a business process BP composed of the processes in PP, there is a process P_x in PP, a $StartEvent$ of P_x is associated with a *message receiver*, receiving the expected message meg, if only if the following conditions hold:

■ $\exists StartEvent \in P_x.StartSet : StartEvent.Timer = None \wedge StartEvent.InMessage = meg \wedge StartEvent.OutMessage = None$

- $\exists P_y \in PP : \widetilde{MF}\left(P_y, P_x\right) = meg$, the $StartEvent$ of P_x receives meg sent from P_y, $P_y \neq P_x$.

An $EndEvent$ of P_x is associated with a *message dispatcher*, submitting message meg, if only if the following conditions hold:

- $\exists EndEvent \in P_x.EndSet : EndEvent.Timer = None \wedge$ $EndEvent.InMessage = None \wedge EndEvent.OutMessage = meg$
- $\exists P_y \in PP : \widetilde{MF}\left(P_x, P_y\right) = meg$, the $EndEvent$ of P_x submits meg to P_y, $P_y \neq P_x$.

An $InterEvent$ of P_x can be associated with a *message receiver* or *dispatcher*. When $InterEvent$ is associated with a message receiver, the following conditions hold:

- $\exists InterEvent \in P_x.InterSet : InterEvent.Timer = None \wedge$ $InterEvent.InMessage = meg \wedge InterEvent.OutMessage = None$
- $\exists P_y \in PP : \widetilde{MF}\left(P_y, P_x\right) = meg$, the $InterEvent$ of P_x receives meg sent from P_y, $P_y \neq P_x$.

When $InterEvent$ is associated with a dispatcher, the following conditions hold:

- $\exists InterEvent \in P_x.InterSet : InterEvent.Timer = None \wedge$ $InterEvent.InMessage = None \wedge InterEvent.OutMessage = Meg$
- $\exists P_y \in PP : \widetilde{MF}\left(P_x, P_y\right) = meg$, the $InterEvent$ of P_x submits meg to P_y, $P_y \neq P_x$.

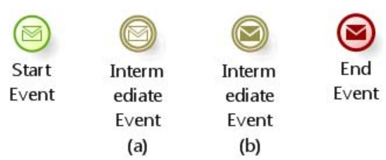


Figure 3.4 The notations for message events.

■ Timing Event

When a process execution reaches an event node which is associated with *timer*, the process is blocked till the time set on the timer. In general, this kind of event is a start or intermediate event, because a process blocked at the end could occupy a resource which other processes are waiting for. Thus, the case is not concerned in our model. When a process is modelled with a *timing start event*, the process can be instantiated at the time (interval) specified. If a process is modelled with a *timing intermediate event*, its execution could be blocked till the time specified or continue within the interval specified. A formal definition of *timing event* is given in Definition 3.9. Notations for the timing events in BPMN are adopted and shown in Figure 3.5.

Definition 3.9. (Timing Event)

Given a process P, a $StartEvent$ of P is associated with *timer* if and only if the following condition holds:

- $\exists StartEvent \in P.StartSet : StartEvent.Timer \neq None \wedge StartEvent.InMessage = None \wedge StartEvent.OutMessage = None$.

An $InterEvent$ of P is associated with *timer* if and only if the following condition holds:

- $\exists InterEvent \in P.InterSet : InterEvent.Timer \neq None \wedge InterEvent.InMessage = None \wedge InterEvent.OutMessage = None$.

Figure 3.5 The notations for the timing events.

In order to describe $Timer$ attribute, we define a time set and a time interval, in a similar formulation as [43]. A time set is a set of all non-negative reals:

$Time = \{x \in REAL \mid x \geq 0\}$. A time interval from x to y is denoted as $[x,y]$, $[x,y] \in Time \times Time$. If $z \in Time$, then $z \in [x,y]$ iff $x \leq z \leq y$. Also, $z \in [x,x]$ iff $x = z$. The set of time interval is defined as $Interval = \{[x,y] \in Time \times Time \mid x \leq y\}$. A formal definition of $Timer$ attribute is given in Definition 3.10.

> **Definition 3.10.** (Timer attribute of Timing Event)
>
> Given a set E, let the timing events of process P be contained in E. The $Timer$ attribute of event in E is defined by $\widetilde{Time}: E \rightarrow Interval$, a timing function maps each timing event to a *static interval* $[min,max]$, which specifies the *earliest start time* and the *latest end time* of event, $min \leq max$. A *dynamic interval* $[\overline{min},\overline{max}]$ is used to denote the active interval of event during an execution.

Given two timing events, u and v, u is a direct predecessor of v and v is set with a static trigger interval $[min,max]$. Let u be triggered at $\tau(u)$ time. The dynamic interval $[\overline{min},\overline{max}]$ of v is shifted by $\tau(u)$: $\overline{min} = Max\{0, \overline{min} - \tau(u)\}$ and $\overline{max} = Max\{0, \overline{max} - \tau(u)\}$. v is allowed to trigger after $\overline{min}$ units of time and should be triggered before $\overline{max}$.

3.2.2 Activities

In a process, an action typed with $Activity$ is a unit of work which makes some function progress. The activity might be atomic or compound. An atomic activity, named as a *task*, is an indecomposable unit of work, while a compound activity contains a group of activities within a process. To be compatible with BPMN, the tasks contained in another task are called the *sub-processes* latter. The set of attributes common to both task and sub-process is defined in Definition 3.11.

Definition 3.11. (Activity)

Given a process P whose control flow is presented by graph $G=(V,CF)$, the activities in set $A=\{v\in V\,|\,\tilde{V}(v)=Activity\}$ have the attributes listed as followings:

1. AT_v is an attribute to represent the type of v, which is defined by $\tilde{A}:A\to T$, a grain function, maps an activity in A into one of the two types in T, where $T=\{Task,SubProcess\}$.
2. Pre_v and Pos_v are the sets of logical expressions which are evaluated by a workflow engine.
   (1) Pre_v is the pre-conditions of which each is evaluated to decide whether activity v within a P instance can be started.
   (2) Pos_v is the post-conditions of which each is evaluated to decide whether activity v is completed.
3. $I_v=IA_v\cup IE_v\cup IM_v$ is a set of inputs, where IA_v identifies all the artifacts required to be accessed by activity v, IE_v is a set of intermediate events could be generated by direct predecessors (activities) for starting an execution of v, and IM_v, $IM_v\subseteq IM_P$, is a set of messages could be received for starting an execution of the corresponding event-driven flow splitting from v or continuing following execution. IE_v and IM_v are defined for constructing event-driven flows.
4. $O_v=OA_v\cup OE_v\cup OM_v$ is a set of outputs, where OA_v identifies all the

artifacts produced, updated or destroyed by v, OE_v contains the events which can be threw out to direct successor from v and OM_v, $OM_v \subseteq OM_P$, is composed of the messages which can be transmitted to other process(es) from v. OE_v and OM_v are defined for constructing event-driven flows.

- $OA_v = OA_v^+ \cup OA_v^-$, where OA_v^+ and OA_v^- are disjoint. OA_v^+ represents the set of artifacts produced or updated by v and OA_v^- represents the set of artifacts destroyed by v.

5. $ST_v(None|Ready|Active|Aborted|Completed)$ represents a state of v during execution. The details are given in Definition 3.13.

6. PF_v defines the resource that performs or is responsible for v, $PF_v = \tilde{P}(P)$.

7. $LT_v = (None|Standard|MultiInstance)$ defines the loop type of activity v. By default, activity v is executed once and the value of LT_v is $None$. $Standard$ and $MultiInstance$ activities are defined in Definition 3.14 and Definition 3.15, respectively.

A process P, created by the basic construction mechanism, contains the activities whose inputs and outputs are artifacts only, i.e., if activity v belongs P, $I_v = IA_v$ and $O_v = OA_v$. When an activity involving event(s) is concerned, the supplement construction mechanism in Definition 3.12 is applied.

Definition 3.12. (Supplement Construction Mechanism).

Given a control flow $G = (V, CF)$, built by the basic construction mechanism, G can be divided into two weakly connected components, $G_u = (V_u, CF_u)$ and $G_v = (V_v, CF_v)$, where $V_u \cap V_v = \phi$ and $CF_u \cap CF_v = \phi$. Let activity u and v belonging to G_u and G_v respectively and $(u,v) \notin CF$ and $InDegree(v) = 0$. When $(|OE_u \cap IE_v| = 1) \wedge (IA_v = \phi)$, *supplement arc* (u,v) can be added into G. $isExtended(u,v)$ is a boolean function to represent if arc (u,v) is added into G.

- $isExtended(u,v) = true \Rightarrow (InDegree(v) = 1) \wedge (|OE_u \cap IE_v| = 1) \wedge (IA_v = \phi)$.

  $isExtended(u,v) = true$ indicates that arc (u,v) is added and activity v is executed when the event et, $et \in (OE_u \cap IE_v)$, involved in u is triggered. $et.ET = Message$ or $et.ET = Time$ can be represented with BPMN as the diagrams shown in Figure 3.6 (a) or (b).

If $IM_u \neq \phi$, $\forall meg \in IM_u$, there is a message inflow (P_x, P) of P, denoted as $\widetilde{MF}(P_x, P) = meg$, $P_x \neq P$. Mapping function $\widetilde{IM}_u : IM_u \rightarrow OE_u$, a one-to-one function, maps each message in IM_u into one of the outgoing events in OE_u, $|OE_u| \geq |IM_u|$.

- When $|OE_u| = |IM_u|$,

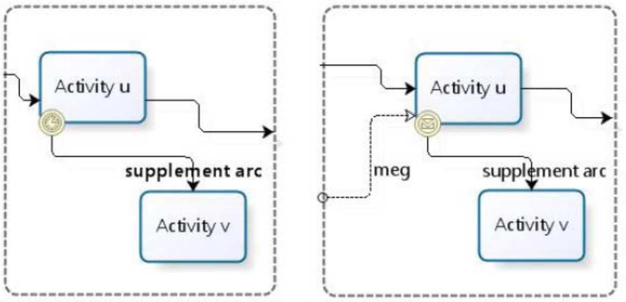
  $OE_u = \{et \mid et.ET = Message \wedge (\exists meg \in IM_u : \widetilde{IM}_u(meg) = et)\}$.

- When $|OE_u| > |IM_u|$,

$$OE_u = \left\{ et \in OE_u \mid et.ET = Message \wedge \left(\exists meg \in IM_u : \widetilde{IM}_u(meg) = et \right) \right\} \cup \left\{ et \in OE_u \mid et.ET = Time \right\}.$$

If $(IM_u = \phi) \wedge (OE_u \neq \phi)$, $OE_u = \{ et \mid et.ET = Time \}$.

In addition, if $OM_u \neq \phi$ as the case shown in Figure 3.7, $\forall meg \in OM_u$, there is a message outflow (P, P_y), denoted as $\widetilde{MF}(P, P_y) = meg$, $P \neq P_y$.



(a) An Activity with a Timing Event (b) An Activity with a Message Receiver

Figure 3.6 The cases of supplement arcs.

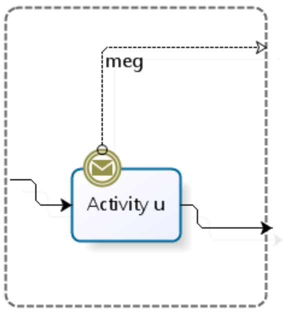


Figure 3.7 An activity with a message dispatcher.

■ Activity States

An activity may change its state when it runs in a workflow engine. In general, there are five process states for an activity inside a process.

1. *None*: an activity has not been admitted to entry the execution pool of workflow engine.
2. *Ready*: an activity does not wait for anything and is prepared to run if it is

selected by workflow engine.

3. *Active*: an activity that is currently being executed.

4. *Aborted*: an activity that cannot be completed because a specified event occurs during its execution.

5. *Completed*: an activity that has been released by workflow engine after a normal termination.

A formal definition of these states is given in Definition 3.13.

Definition 3.13. (States of Activity).

For a given activity v, the state ST of an instance of v can be defined by its incoming and outgoing data (artifacts, events and messages) and the input and output set specified, $I_v = IA_v \cup IE_v \cup IM_v$ and $O_v = OA_v \cup OE_v \cup OM_v$. The default value of ST_v is $None$.

$$ST_v = (None \mid Ready \mid Active \mid Aborted \mid Completed)$$

Let $\hat{I}_v = \widehat{IA}_v \cup \widehat{IE}_v \cup \widehat{IM}_v$ be a set of inputs received by v at run time, where $\widehat{IA}_v$ contains the artifacts propagated from the predecessor(s) directly connected by data flow(s), $\widehat{IE}_v$ contains the events received from the preceding activity connected by supplement arc(s) and $\widehat{IM}_v$ contains the messages received from the preceding action(s) connected by message flows.

Let $\hat{O}_v = \widehat{OA}_v \cup \widehat{OE}_v \cup \widehat{OM}_v$ be a set of outputs submitted from v at run time, where $\widehat{OA}_v$ contains the artifacts propagated to the successor(s) directly connected by data flow(s), $\widehat{OE}_v$ contains the events submitted to the succeeding activity connected by supplement arc(s) and $\widehat{OM}_v$ contains the messages submitted to the succeeding activity(ies) and/or intermediate message event(s) connected by message flows. In addition, $\widehat{OA}_v = \widehat{OA}_v^- \cup \widehat{OA}_v^+$, where

$\widehat{OA}_v^{+}$ and $\widehat{OA}_v^{-}$ represents the sets of artifacts produced/updated and destroyed, respectively. All the possible states of v are defined as follows:

When $I_v = IA_v$ and $O_v = OA_v$,

- If $(IA_v = \phi) \wedge (OA_v \neq \phi)$, the default state of v is $Ready$.
- If $OA_v \setminus \widehat{OA}_v = \phi$, the state of v is $Completed$.
- If $(IA_v \neq \phi) \wedge (OA_v = \phi)$, the default state of v is $None$.
  - Once $IA_v \setminus \widehat{IA}_v = \phi$, ST_v is transferred from $None$ to $Ready$.
- If $(IA_v \neq \phi) \wedge (OA_v \neq \phi)$, the default state of v is $None$.
  - Once $(IA_v \setminus \widehat{IA}_v = \phi) \wedge (OA_v \setminus \widehat{OA}_v = OA_v)$, ST_v is transferred from $None$ to $Ready$.
  - Once $(IA_v \setminus \widehat{IA}_v = \phi) \wedge (OA_v \setminus \widehat{OA}_v \subset OA_v)$, the state of v is $Active$.
  - Once $(IA_v \setminus \widehat{IA}_v = \phi) \wedge (OA_v \setminus \widehat{OA}_v = \phi)$, the state of v is $Completed$.

If $OM_v \neq \phi$, the messages defined in OM_v are submitted when the state of v is $Completed$. In addition, if there is an activity u, a direct predecessor of v connected by supplement arc, $|OE_u \cap IE_v| = 1$, $ST_u = Active$ and $ST_v = Ready$.

- When $IM_u = \phi$,
  - if $((OE_u \cap IE_v) \setminus \widehat{OE}_u = \phi)$, ST_u is transferred from $Active$ to $Aborted$ and ST_v is transferred from $Ready$ to $Active$.

- When $IM_u \neq \phi$ and $\exists meg \in IM_u : \widetilde{IM}_u(meg) \in (OE_u \cap IE_v)$,
  - if $\left(\exists meg \in \widehat{IM}_u : \widetilde{IM}_u(meg) \in (OE_u \cap IE_v)\right) \wedge \left((OE_u \cap IE_v) \backslash \widehat{OE}_u = \phi\right)$, ST_u is transferred from *Active* to *Aborted* and ST_v is transferred from *Ready* to *Active*.
- When $IM_u \neq \phi$ and $\nexists meg \in IM_u : \widetilde{IM}_u(meg) \in (OE_u \cap IE_v)$,
  - if $\left((OE_u \cap IE_v) \backslash \widehat{OE}_u = \phi\right)$, ST_u is transferred from *Active* to *Aborted* and ST_v is transferred from *Ready* to *Active*.

- **Loop Activity**

There are three different loops of activity, *None*, *Standard* and *MultiInstance*. None-loop activities are executed once only. Except such activities, the execution times of activities implemented with the remaining two types are decided by the expression evaluation results.

There are two standard loop for activities: *While* and *RepeatUntil*. The expressions associated with these loops return with boolean value. A While loop evaluates the expression before the activity is performed. A RepeatUntil loop evaluates the expression after the activity has been performed. Obviously, the least time of *RepeatUntil* (R)/ *While* (W) execution is 1/0. During an execution, the number of iterations is bounded and recorded. These specific attributes of standard loop activity are defined in Definition 3.14.

The numeric expression for an activity, designed with *MultiInstance* loop, is evaluated once only before the activity is performed. The evaluation result is an integer that specifies the number of times that the activity will be repeated. There are two variations of the multi-instance loop where the instances are either performed sequentially or in parallel. When a multi-instance loop is performed in parallel, the execution of these instances can be categorized into three cases: (1) all instances continue to execute succeeding flow when that instance is completed, (2) all instances continue to execute succeeding flow after one of the instances is completed and (3) all instances continue to execute succeeding flow after all of the instances are completed. In case (1), the number of instances available for the succeeding flow of activity v is the same as the number of v's instances. In case (2) and (3), there is only one instance available for the succeeding flow. Thus, the number of the instances which will be available for the continuing flow is determined by the way adopted. The specific attributes of multi-instance loop are defined in Definition 3.15.

Definition 3.14. (Attributes of Standard Loop Activity)

Given an activity v, when $LT_v = Standard$, v has some additional attributes listed followings:

1. $BooleanExp_v$ is the set of routing conditions of which each is evaluated before or after the execution of v,
2. $Counter_v$ is an integer used at run time to record the number of iterations executed,
3. $Maximum_v$ is an finite integer by which the number of loops executed is bounded, $Maximum_v \geq Counter_v$,
4. $EvTime_v = (Before \mid After)$ attribute denotes that $BooleanExp_v$ is evaluated before or after the execution of v.

Definition 3.15. (Attributes of Multi Instance Loop Activity)

Given an activity v, when $LT_v = MultiInstance$, v has some additional attributes listed followings:

1. $NumExp_v$ is a numeric expression to decide the number of instances of v.
2. $Order_v = (Sequential \mid Parallel)$ attribute denotes the instances of v are performed sequentially or in parallel.
3. $Counter_v$ is an integer and only applied for v whose instances are performed sequentially. The integer is used at run time to record the number of iterations executed.
4. $FlowCond_v = (None \mid One \mid All)$ attribute sets the way of controlling the instances of v executed in parallel.
   (1). When $FlowCond_v = None$, all instances of v continue to execute succeeding flow when that instance is completed.
   (2). When $FlowCond_v = One$, all instances of v continue to execute succeeding flow after one of the instances is completed
   (3). When $FlowCond_v = All$, all instances of v continue to execute succeeding flow after all of the instances are completed

Notations for loop activity in BPMN are adopted and shown in Figure 3.8.

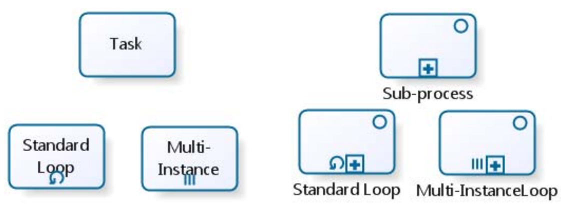


Figure 3.8 Notations for loop activities.

3.2.3 Control Nodes

In a process, an action v typed with $ControlNode$ is associated with a mechanism which is used to control how the activities interact as they converge and diverge within a process. A formal definition of control node is given in Definition 3.16.

Definition 3.16. (Control Node)

Given a process P whose control flow is $ControlFlow(P)$ presented by graph $G=(V,CF)$, the control notes in set $C=\{v\in V\,|\,\tilde{V}(v)=ControlNode\}$ have the attributes listed as followings:

1. CT_v is an attribute to present the control mechanism of v which is defined by $\widetilde{CT}:C\rightarrow T$, a type function maps each activity in C into one of the four types of control mechanism in T, where $T=\{Exclusive,Inclusive,Complex,Parallel\}$.
2. IA_v is a set identifying all the artifacts required to be accessed by v.

A group of actions can be bounded by a pair of control nodes. Each pair and the actions bounded by them are called *control block*. Given a process built by basic construction mechanism, the structure of the process is sequential, when no control node is included. Otherwise, there may be control blocks in the process. When any two of control blocks in the process, B_1 and B_2, are nested but not overlap, $(B_1\subset B_2)\vee(B_2\subset B_1)$, the level of an action belonging to either blocks, applied for the followings, can be defined as the definitions given in Definition 3.17 and Definition 3.18.

Definition 3.17. (Ancestor Blocks and Level of an Action)

$\forall v \in V$, let $v.Block$ denote the parent control block containing v. $\overline{AncestorBlock}$ comprises the set of all control blocks that contain v.

$\overline{AncestorBlock}(v) = \{b \mid b = v.Block \vee (b \in \overline{AncestorBlock}(v.Block.splitNode)\}$

In addition, the cardinality of $\overline{AncestorBlock}(v)$ identifies the nested level of v.

$$Level(v) = \begin{cases} \left|\overline{AncestorBlock}(v)\right| & \text{if } v \in V \\ \left|\overline{AncestorBlock}(v.splitNode)\right| & \text{if } v \text{ represents a control block} \end{cases}$$

Definition 3.18. (Common Ancestor Blocks and Nearest Common Ancestor Blocks)

Given a set of vertices, $v_1, \ldots, v_n$, B_i is a *common ancestor block* of $v_1, \ldots, v_n$ if and only if the following holds:

$B_i \in \bigcap_{i=1}^{n} \overline{AncestorBlock}(v_i)$, denoted by $B_i \in CAB(v_1, \ldots, v_n)$.

B_i is the *Nearest common ancestor* of $v_1, \ldots, v_n$ if and only if the following holds: $\forall B_j \in CAB(v_1, \ldots, v_n) \wedge B_j \neq B_i : Level(B_i) < Level(B_j)$, denoted by $NCAB(v_1, \ldots, v_n) = B_i$.

When a control node v is constructed with one incoming edge and more than one outgoing edge, $InDegree(v) = 1 \wedge OutDegree(v) > 1$, v is named as *split node*. Otherwise, v is called *join node*, constructed with more than one incoming edge and one outgoing edge, $InDegree(v) > 1 \wedge OutDegree(v) = 1$. There are four different mechanisms, $Exclusive$, $Inclusive$, $Complex$ and $Parallel$, defined in our model. Except the $Complex$ mechanism, the remains can be pairwise applied on split and join nodes. The $Complex$ mechanism can be applied on join node only. Upon the ways of adopting mechanism(s), we divide the control blocks developed into two groups: *fundamental* and *complex*. In the fundamental group, the control blocks, *exclusive*, *inclusive*, and *parallel*, are bounded with split and join nodes designed with the same mechanism. Formulations of these four types of blocks are given in Definition 3.19, Definition 3.20, and Definition 3.21, respectively.

Definition 3.19. (Exclusive Control Block)

Given a process P whose control flow is $ControlFlow(P)$ presented by graph $G=(V,CF)$, there is a exclusive control block (v,k) in P, such that $v,k \in V:(\tilde{V}(v)=\tilde{V}(k)=ControlNode)\wedge(Level(v)=Level(k))$ and $(v.CT=ExclusiveSplit)\wedge(k.CT=ExclusiveJoin)$.

During an execution, v takes one of its outgoing flows to continue upon one of the two sources: *data-based* and *event-based*.

- v is a $DataBasedExclusiveSplit$ node if and only if v is associated with an expression $ChoiceExp$ which is evaluated by using the data propagated from direct data-flow predecessor(s). Besides, $v.IA \neq \phi$.
- v is an $EventBasedExclusiveSplit$ node if and only if

$$\forall u \in V_v^{IsSuccessor} \mid \begin{pmatrix} (\tilde{A}(u)=Task \wedge IM_u \neq \phi)\vee \\ (u \in P.InterSet : u.Timer \neq None \vee u.InMessage \neq None) \end{pmatrix}$$

and the outflow selected to run is the one whose event occurs first. Besides, $v.IA=\phi$.

The outgoing flows of either $DataBasedExclusiveSplit$ or $EventBasedExclusiveSplit$ node are merged at $DataBasedExclusiveJoin$ node

k . The following process is continued through the execution reaches $DataBasedExclusiveJoin$ node.

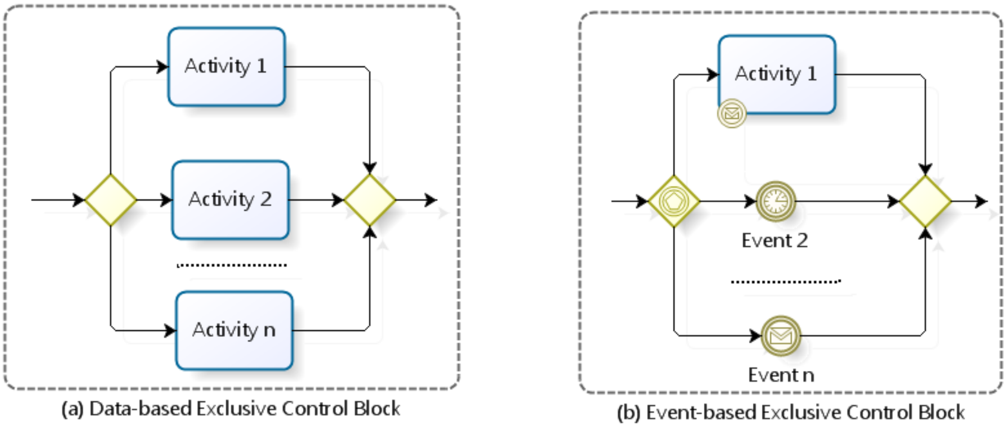


Figure 3.9 Samples of exclusive control block.

Definition 3.20. (Inclusive Control Block)

Given a process P whose control flow is $ControlFlow(P)$ presented by graph $G=(V,CF)$, there is an inclusive control block (v,k) in P, such that $v,k \in V:\left(\tilde{V}(v)=\tilde{V}(k)=ControlNode\right)\wedge\left(Level(v)=Level(k)\right)$ and $(v.CT=InclusiveSplit)\wedge(k.CT=InclusiveJoin)$.

For v, an $InclusiveSplit$ node, one to all of its outgoing flows are selected to run. The number of executive outflows is determined by the expression $ChoiceExp$ associated with v, which is evaluated by the data propagated from direct predecessor(s), $v.IA \neq \phi$, connected by data flow(s).

For k, an $InclusiveJoin$ node, is used to synchronize all the executive branches before continuing to the next action.

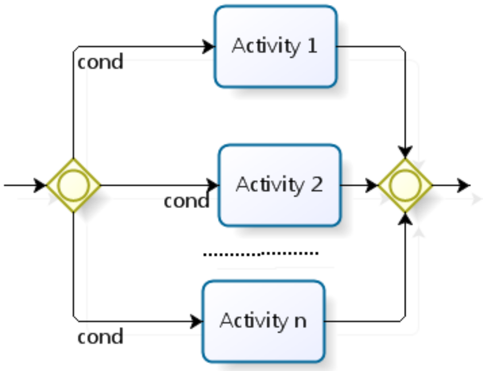


Figure 3.10 A sample of inclusive control block.

Definition 3.21. (Parallel Control Block)

Given a process P whose control flow is $ControlFlow(P)$ presented by graph $G=(V,CF)$, there is a parallel control block (v,k) in P, such that $v,k\in V:\tilde{V}(v)=\tilde{V}(k)=ControlNode\wedge Level(v)=Level(k)$ and $(v.CT=ParallelSplit)\wedge(k.CT=ParallelJoin)$.

For v, a $ParallelSplit$ node, all its outgoing flows are selected to run and k, an $ParallelJoin$ node, is used to synchronize all these executive flows before continuing to the next action.

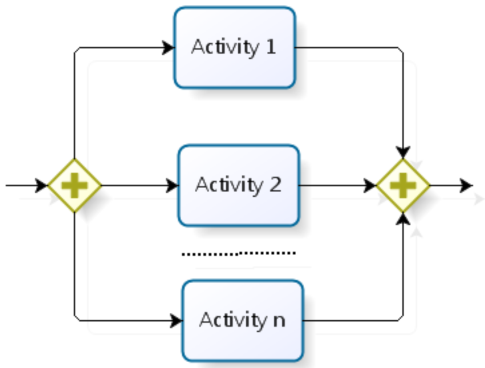


Figure 3.11 A sample of parallel control block.

A loop is bounded by two $DataBasedExclusive$ nodes as the samples shown in

Figure 3.12. The actions bounded within the two nodes can be executed repeatedly based on a given boolean condition. The pair of control nodes and these repeated actions bounded by them are called *loop control block*, defined in Definition 3.22.

Definition 3.22. (Loop Control Block)

Given a process P whose control flow is $ControlFlow(P)$ presented by graph $G=(V,CF)$, there is an loop control block (v,k) in P, such that

- when v is associated with a boolean expression $BooleanExp$, which is evaluated before each iteration, the control block is called $WhileLoop$ control block.

$$v,k \in V: \left(\tilde{V}(v)=\tilde{V}(k)=ControlNode\right) \wedge \left(Level(v)=Level(k)\right)$$

$$\left(v.CT=Exclusive\right) \wedge \left(InDegree(v)=2 \wedge OutDegree(v)=2\right)$$

$$\left(k.CT=Exclusive\right) \wedge \left(InDegree(v)=2 \wedge OutDegree(v)=2\right).$$

- when k is associated with a boolean expression $BooleanExp$, which is evaluated after each iteration, the structure is called $RepeatUntilLoop$ control block.

$$v,k \in V: \tilde{V}(v)=\tilde{V}(k)=ControlNode \wedge Level(v)=Level(k)$$

$$\left(v.CT=Exclusive\right) \wedge \left(InDegree(v)=2 \wedge OutDegree(v)=1\right)$$

$$\left(k.CT=Exclusive\right) \wedge \left(InDegree(v)=1 \wedge OutDegree(v)=2\right).$$

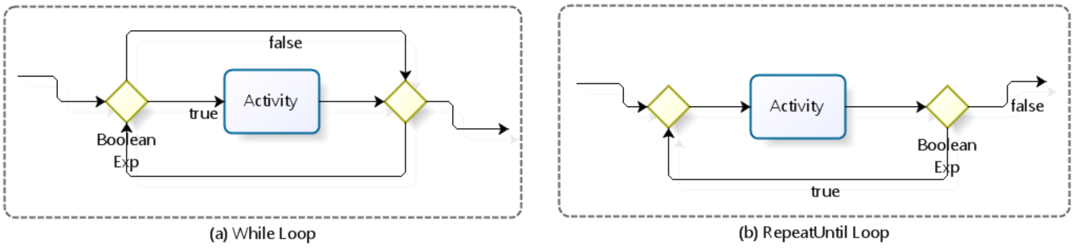


Figure 3.12 Samples of loop control block.

In addition to the fundamental blocks, the complex blocks, bounded with two control nodes associated with different mechanisms, are included in our model. In such block, the flows are split from either *InclusiveSplit* or *ParallelSplit* node and joined at a *ComplexJoin* node. There are three advanced join mechanisms, *discriminator* , *Multiple Merge* and *N out of M join* proposed in [36], which can be implemented with the *ComplexJoin* node. The details of these advanced mechanisms are described as followings:

1. Discriminator

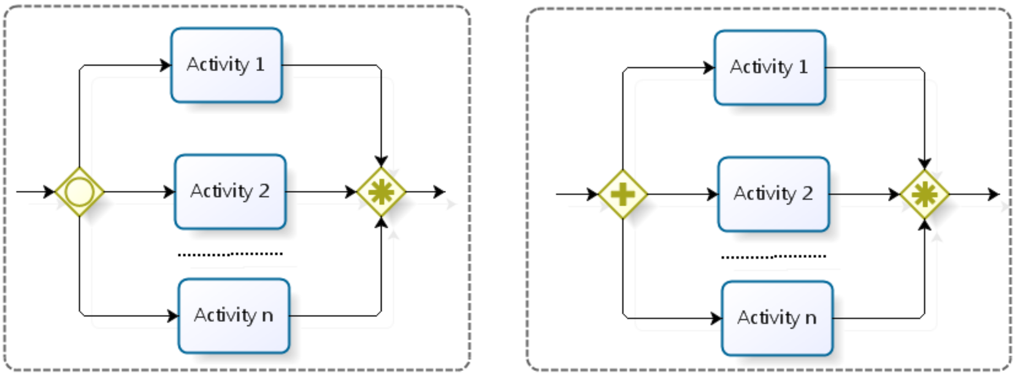
The *ComplexJoin* node continues to execute the following flow when one of its inflows is completed. The remaining inflows are excluded, even they are completed later.

2. Multiple Merge

Each inflow of the *ComplexJoin* node continues to execute succeeding flow when that flow is completed.

3. N out of M join

The *ComplexJoin* node associated with an expression which is evaluated to synchronize the first M incoming flows from N executive inflows, $N \geq M$.



(a) A complex control block built with a inclusive split node. (b) A complex control block built with a parallel split node.

Figure 3.13 The samples of complex blocks.

The twelve control blocks concerned in this thesis are listed in Table 3.1. We assume that the specific correlations between the two control nodes of these blocks are maintained.

Table 3.1 Control blocks

Control Block	**Split Control Node**	**Join Control Node**
DataExclusive	*DataBasedExclusiveSplit*	*DataBasedExclusiveJoin*
EventExclusive	*EventBasedExclusiveSplit*	*DataBasedExclusiveJoin*
Inclusive	*InclusiveSplit*	*InclusiveJoin*
Parallel	*ParallelSplit*	*ParallelJoin*
WhileLoop	*DataBasedExclusive*	*DataBasedExclusive*
RepeatUntilLoop	*DataBasedExclusive*	*DataBasedExclusive*
ParallelDiscriminator	*ParallelSplit*	*ComplexJoin* (Discriminator)
InclusiveDiscriminator	*InclusiveSplit*	*ComplexJoin* (Discriminator)
ParallelMultiMerge	*ParallelSplit*	*ComplexJoin* (Multiple Merge)
InclusiveMultiMerge	*InclusiveSplit*	*ComplexJoin* (Multiple Merge)

ParallelNtoM	*ParallelSplit*	*ComplexJoin* (*N* out of *M* join)
InclusiveNtoM	*InclusiveSplit*	*ComplexJoin* (*N* out of *M* join)

3.2.4 A Control Flow Example: a Process of Resolving Issues through E-mail Votes

The message flows in Figure 3.2 indicates that e-mail voting process BP_{vote} is divided into three private processes, $P_{workingGroup}$, $P_{manager}$ and P_{voter}. Our control flow model is then adopted to construct the details of these private processes from a view point of process control, i.e., the actions assigned to the three involving roles, working group, manager and voter, are defined and shown in Figure 3.14.

BP_{vote} has turn cycle of a week. Private process $P_{workingGroup}$ is instantiated at 9 in the morning on each Monday. First of all, the working group involved checks its status. If the status of the group is inactive, the process instance is terminated. Otherwise, the issues raised in the group are listed and a manager is notified. Process $P_{manager}$, instantiated with the notification and the manager, responsible for the process instance, reviews these issues proposed. The review results are announced to voting members, respectively. Each announcement instantiates a P_{voter} process with one voting member and the process has to complete its activity before Friday.

Manager collects votes through executing sub-process $SP_{2.1}$ whose detail flow is shown in Figure 3.15, where there are three control blocks, $Parallel(PS2.1.1, PJ2.1.1)$, $DataExclusive(DaES2.1.1, EJ2.1.1)$ and $WhileLoop(EvE2.1.1, DaE2.1.1)$. The latter two control blocks are located in two different branches of the control block $Parallel(PS2.1.1, PJ2.1.1)$.

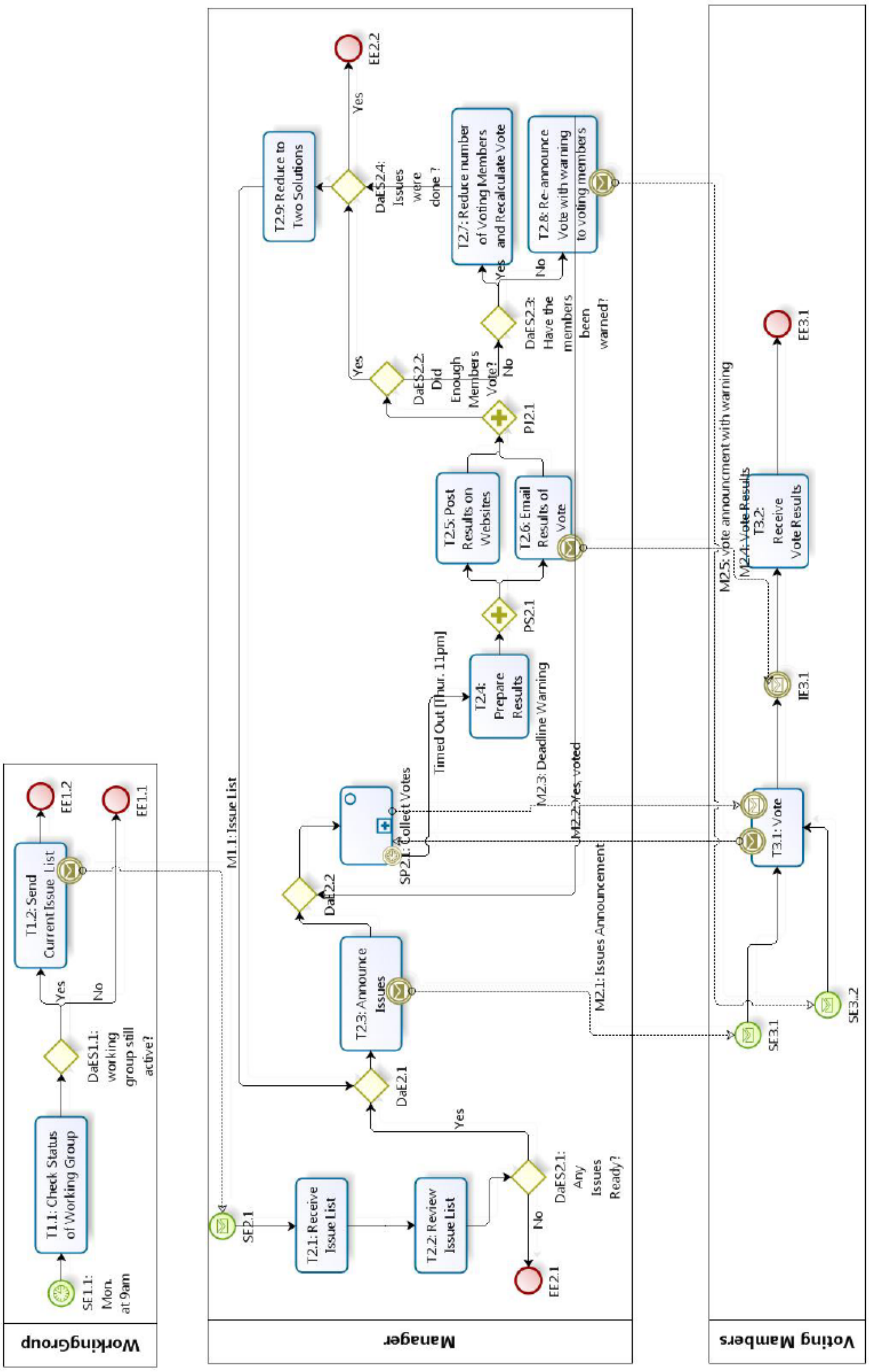


Figure 3.14 The control flow of the business process for resolving issues.

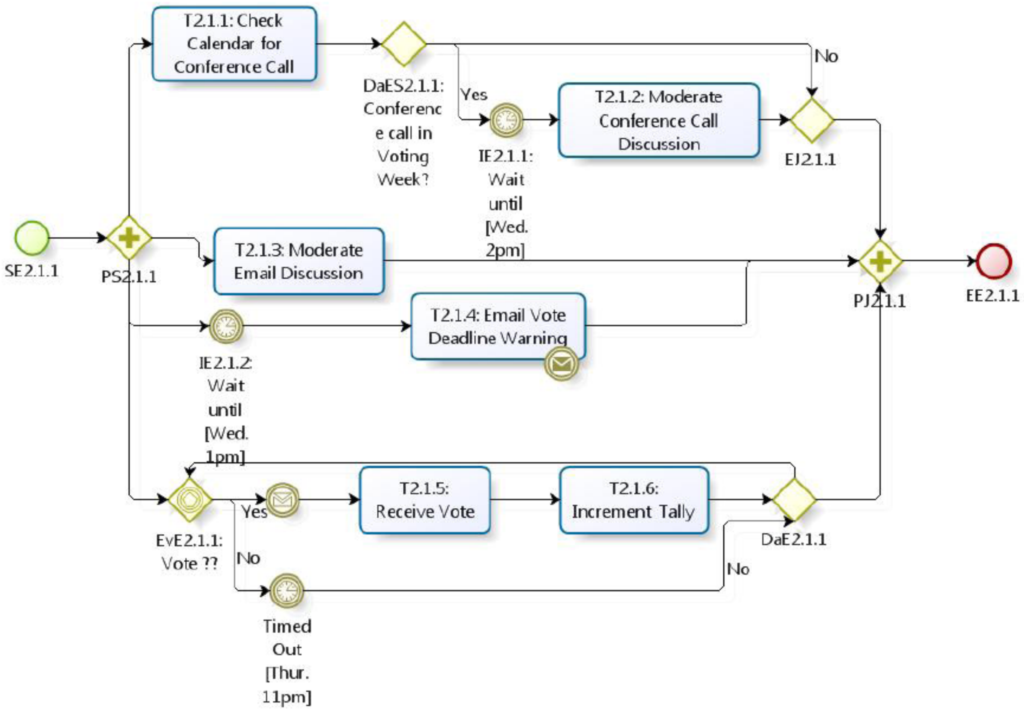


Figure 3.15 The expansion of Collect Vote sub-process.

The manager reports the voting results to voting members when the timing event involved in $SP_{2.1}$ occurs, i.e., the supplement flow of $SP_{2.1}$ is executed. When the number of votes is more than the number specified on the condition of $DaES_{2.2}$, and all the issues listed are done by working group, the instance of $P_{manager}$ terminates. Otherwise, such as insufficient votes, the manager re-announces the vote with warning to the voting member(s) who has not vote in the restricted interval. For the unsolved issues, the manager reduces the number of choices to two and re-announces the vote to the voting members. These two cases are respectively handled by the actions bounded within two pair of control nodes, $(DaES2.4, DaE2.1)$ and $(DaES2.3, DaE2.1)$. The above actions execute repeatedly until the conditions associated with $DaES2.4$ and $DaES2.3$ are satisfied.

3.3. Data Flow Specification

3.3.1. Artifacts and Artifact Operations

Artifacts are the information entities involved in a process, including the input data to the process, the intermediate data produced within the process, and the (final) output data from the process. An artifact is an atomic data item (e.g. a number, a character string, or an image) or a collection of atomic data items (e.g. a document). Intuitively, all artifacts participating in a workflow execution must be pre-defined in a process specification. Each artifact contains a set of legal operations for its internal data. A data-based action designed to manipulate certain artifact can work only with the legal operation(s) for the artifact. From the data storage point of view, each artifact operation can be regarded as one of the following operations, regardless of its semantic meaning:

1. *Initialize*: an operation that instantiates artifact(s) within a process.
2. *Read/Update/Destroy*: an operation that refers/modifies/deletes the artifact instance(s) propagated from predecessor(s) or contained in input data only.

In general, an *Initialize* operation is used to create an artifact instance in a process. *Read* and *Update* operations are then used to access the instance. Finally, a *Destroy* operation is used to delete the artifact instance. *Destroy* operations are applied for temporary artifacts created during the workflow execution, but may not be strict for all artifacts.

Figure 3.16 shows the state transition diagram of an artifact with the above four kinds of operations. 'Uninitialized' represents the initial state of an artifact. "Initialized", "Updated", and "Read" represent states after an *Initialize, Update*, and *Read* operation is performed respectively. In addition, the artifact state is set to "Uninitialized" after a *Destroy* operation.

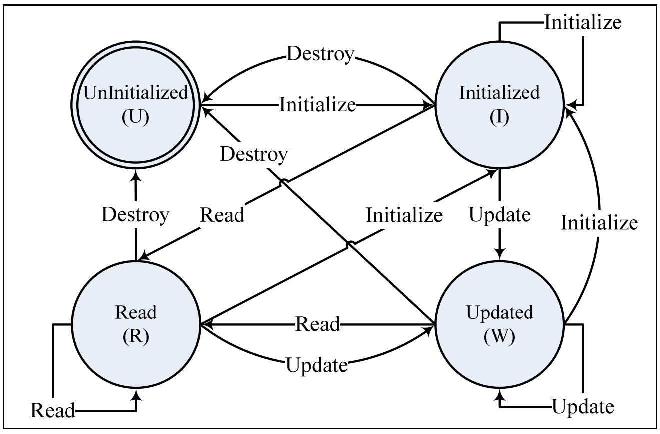


Figure 3.16 The state transition diagram of an artifact.

3.3.2. Artifact Usages

Based on Definition 3.11, a usage relation between a data-based action and an artifact can be defined as follows:

Definition 3.23. (Consumer, Producer, Updator, and Destroyer Actions of an Artifact)

For a given artifact *d*, the memberships between artifact *d* and I_v, O_v^+, and O_v^- can be applied for identifying the usage of artifact *d* at action *v*. All the possible usages are categorized as follows:

- if $d \in IA_v$ and $\begin{cases} d \notin OA_v^+ \\ d \notin OA_v^- \end{cases}$, *v* is called a *Reader*(Action) of artifact *d*.
- if $d \in IA_v$ and $d \in OA_v^+$, *v* is called an *Updator*(Action) of artifact *d*.
- if $d \in IA_v$ and $d \in OA_v^-$, *v* is called a *Destroyer*(Action) of artifact *d*.
- if $d \notin IA_v$ and $d \in OA_v^-$, *v* is called a *Illegal Destroyer*[5](Action) of artifact *d*.
- if $d \notin IA_v$ and $d \in OA_v^+$, *v* is called a *Producer*(Action) of artifact *d*.

[5] The illegal destroyer is not concerned in our model because the activity destroy artifact arbitrarily. Any useful artifact could be destroyed by the activity during the workflow execution.

■ if $d \notin IA_v$ and $\begin{cases} d \notin OA_v^+ \\ d \notin OA_v^- \end{cases}$, *v* is called an *Irrelevantor*(Action) of artifact *d*.

In addition, if $d \in IA_v$, *v* is generally called a *Consumer*(Action) of artifact *d* and if $d \in OA_v^+$, *v* is generally called a *Writer*(Action) of artifact *d*.

Definition 3.24. (Consumer, Writer, Updator, Destroyer, Producer and Reader Action Sets of an Artifact).

■ $V_d^{IsConsumer} = \{v \in V \mid d \in IA_v\}$ is called the *Consumer Action Set* of artifact *d*.

■ $V_d^{IsWriter} = \{v \in V \mid d \in OA_v^+\}$ is called the *Writer Action Set* of artifact *d*.

■ $V_d^{IsUpdator} = \{v \in V \mid d \in IA_v \text{ and } d \in OA_v^+\}$ is called the *Updator Action Set* of artifact *d*.

■ $V_d^{IsDestroyer} = \{v \in V \mid d \in IA_v \text{ and } d \in OA_v^-\}$ is called the *Destroyer Action Set* of artifact *d*.

■ $V_d^{IsProducer} = \{v \in V \mid d \notin IA_v \text{ and } d \in OA_v^+\}$ is called the *Producer Action Set* of artifact *d*.

■ $V_d^{IsReader} = \{v \in V \mid d \in IA_v, d \notin OA_v^+ \text{ and } d \notin OA_v^-\}$ is called the *Reader Action Set* of artifact *d*.

3.3.3. Definition of Data Flow

There are three artifact transmission models identified by Aalst in [37], which are: (1) global data store, (2) integrated control and data channels, and (3) distinct control and data channels. The model implemented with distinct control and data channels is an easier way to represent the transmission of authorized artifacts [44]. Artifacts are transmitted from a data-based action to its following action(s). The transmissions are represented with data flows, defined in Definition 3.25.

Definition 3.25. (Data Flow Specification)

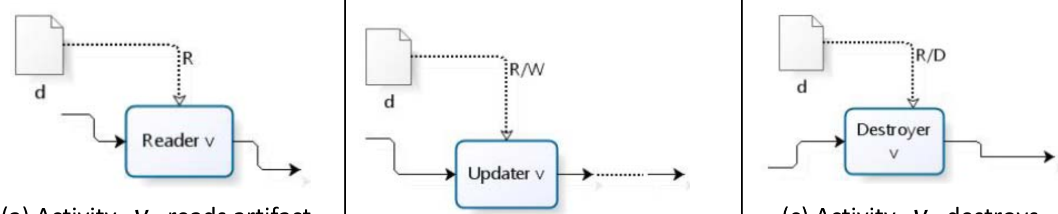
For a given business process BP, one of its private process P is associated with $ControlFlow(P)=\left(G,\tilde{V},A,M,I,O\right)$ where $G=(V,CF)$.

The data flow associated with P is specified with $DataFlow(P)=InDataFlow(P)\cup InterDataFlow(P)\cup OutDataFlow(P)$, where

- $InDataFlow(P)=\left\{(d,v)\in IA\times V \mid v\in V_d^{IsConsumer}\right\}$ is a set of *incoming data flows* where an element (d,v) denotes the inputted artifact d, $d\in I$, consumed by v.
- $InterDataFlow(P)=\left\{\left((u,v),d\right)\in(V\times V)\times A \mid v\in \overline{V_u^{IsSuccessor}}\cap V_d^{IsConsumer}\right\}$ is a set of *intermediate data flows* where an element $\left((u,v),d\right)$ presented by a directed edge to indicate artifact d sent from u to consumer v, a successor of u.

  When there is no incoming data flow of u indicating artifact d sent from preceding action or included in process artifact inputs, u is a producer of artifact d. Otherwise, u consumes artifact d before sending and delegating the access right of d to v.
- $OutDataFlow(P)=\left\{(v,d)\in V\times OA \mid v\in V_d^{IsWriter}\right\}$ is a set of *outgoing data flows* where an element (v,d) denotes process output d contributed from v.

For incoming data flow (d,v), the artifact input d can be read, updated or destroyed by activity v. The three cases of incoming data flows are presented as that shown in Figure 3.17 (a), (b) and (c), respectively.



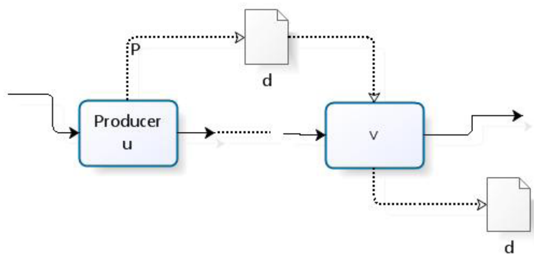
(a) Activity v reads artifact input d

(b) Activity v updates artifact input d

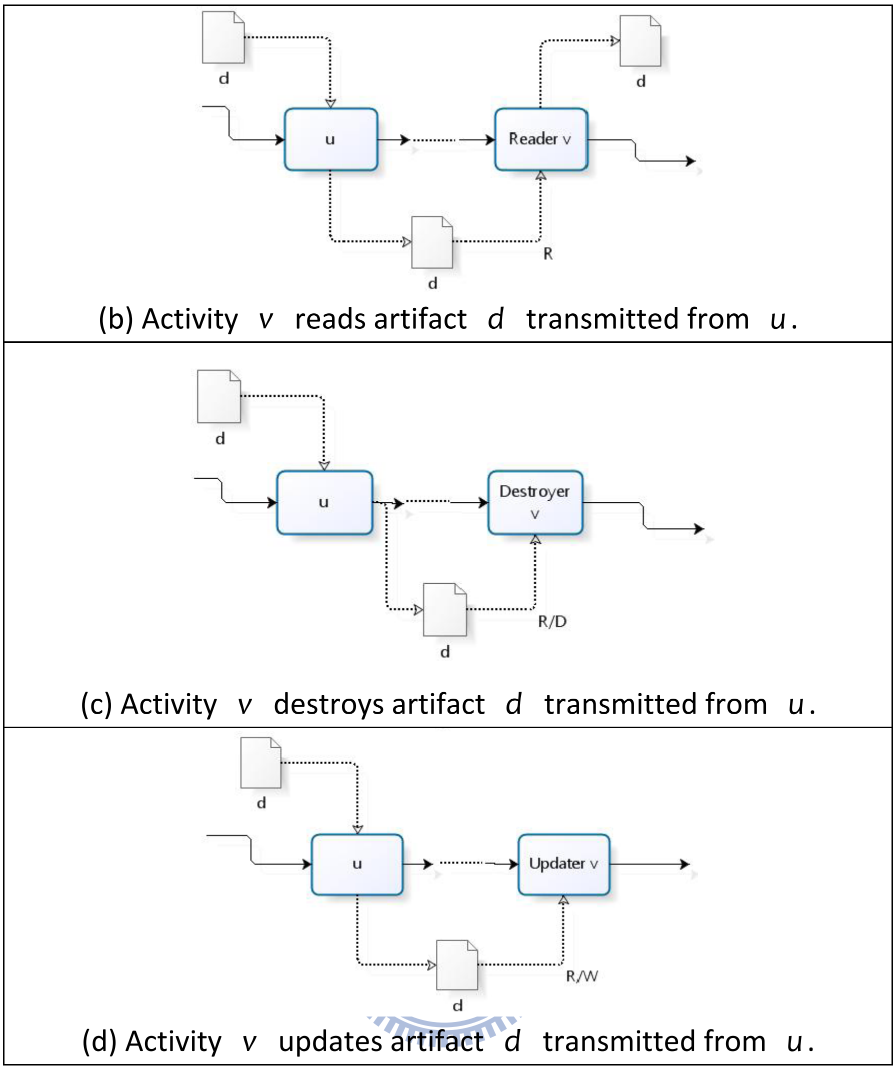
(c) Activity v destroys artifact input d

Figure 3.17 Three cases of incoming data flows.

For intermediate data flow $((u,v),d)$, the artifact d is either produced by or transmitted from action u, such as the two examples shown in Figure 3.18 (a) and (b), to consumer v. v could read, destroy or update artifact d propagated from u. The graphical presentations of the three consuming operations are shown in Figure 3.18 (b), (c) and (d), respectively. In addition, outgoing data flow (v,d) can be presented as the examples shown in Figure 3.18 (a), where process output d is contributed from v.



(a) Activity u produces artifact d.



(b) Activity v reads artifact d transmitted from u.

(c) Activity v destroys artifact d transmitted from u.

(d) Activity v updates artifact d transmitted from u.

Figure 3.18 The four cases of intermediate data flows.

3.3.4. A Data Flow Example: a Process of Resolving Issues through E-mail Votes

Our data flow model is applied on the control flows of e-mail voting process BP_{vote}, shown in Figure 3.14, to illustrate the steps to present the data transformations within BP_{vote}. Figure 3.19 shows the result of representing business process BP_{vote} with both control and data flows. The artifacts in BP_{vote} are stated with details in Table 3.2. The artifact usages in the actions are listed in Table 3.3.

Table 3.2 Artifacts in the E-mail Voting Process

Artifacts			
d_1	Issue List	d_4	Voting Tally
d_2	Vote	d_5	Voting Results
d_3	Calendar		

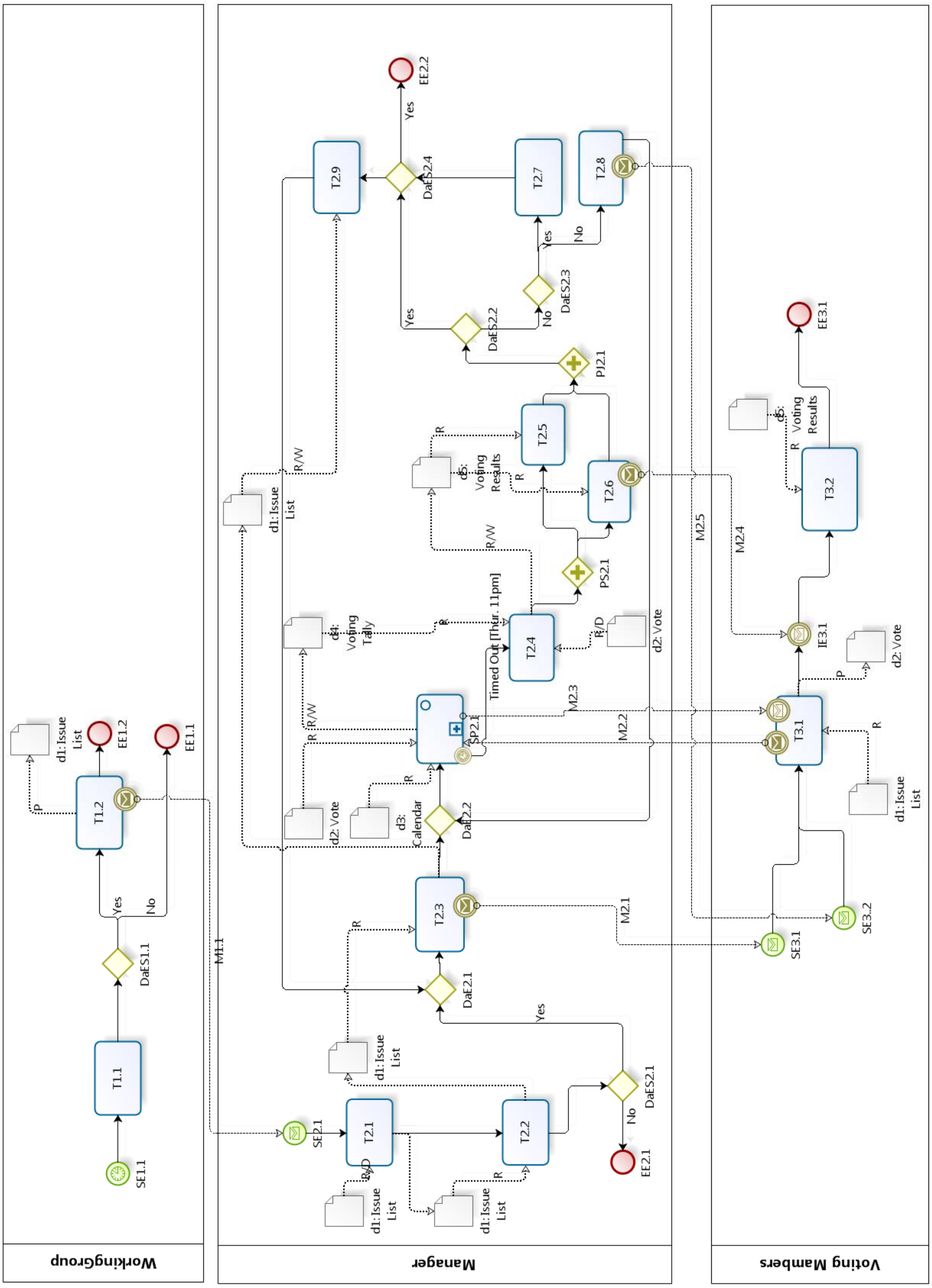


Figure 3.19 The control and data flows of BP_{vote}.

Table 3.3 Artifacts Usages in the E-mail Voting Process

Action	d1	d2	d3	d4	d5
T1.1 Check Status of Working Group					
T1.2 Send Current Issue List	P				
T2.1 Receive Issue List	D				
T2.2 Review Issue List	R				
T2.3 Announce Issues	R				
T2.4 Prepare Results		D		R	U
T2.5 Post Results on Websites					R
T2.6 Email Results of Vote					R
T2.7 Reduce Number of Voting Members and Recalculate Vote					
T2.8 Re-announce Vote with Warning to Voting Members					
T2.9 Reduce to Two Solutions	U				
SP2.1 Collect Votes		R	R	U	
T2.1.1 Check Calendar for Conference Call			R		
T2.1.2 Moderate Conference Call Discussion					
T2.1.3 Moderate Email Discussion					
T2.1.4 Email Vote Deadline Warning					
T2.1.5 Receive Vote		R			
T2.1.6 Increment Tally		R		U	
T3.1 Vote		P			
T3.2 Receive Vote Results					R
R Reader **U** Updater **P** Producer **D** Destroyer					

For the expansion of sub-process $SP2.1$ “Collect Votes”, shown in Figure 3.20, there are two incoming data flows, $(d3, T2.1.1)$ and $(d2, T2.1.5)$, one intermediate data flow $((T2.1.5, T2.1.6), d2)$ and one outgoing data flow $(T2.1.6, d4)$. For the incoming data flow $(d3, T2.1.1)$, manager executes task $T2.1.1$ by referring the input calendar $d3$ to make a conference call. Except incoming data flow

$(d3, T2.1.1)$, the remaining data flows are bounded within $WhileLoop$ control block $(EvE2.1.1, DaE2.1.1)$. Manager refers the voting data $d2$ received to update the vote tally $d4$ recursively till the time limitation denoted is arrived. The final version of $d4$ contributes to process output.

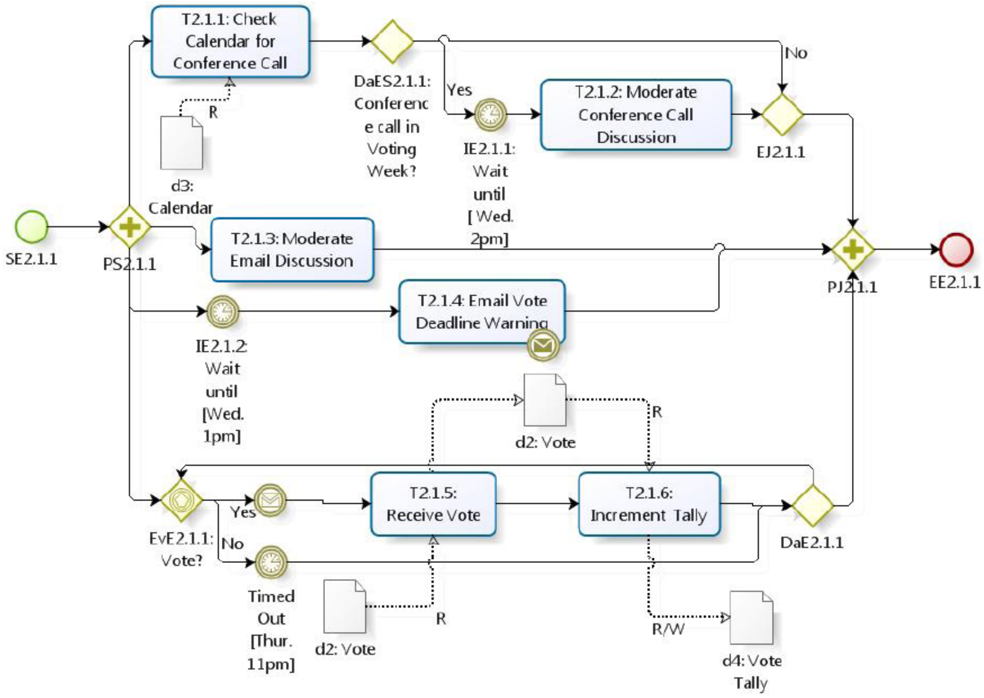


Figure 3.20 The expansion of "Collect Votes" sub-process.

3.3.5. Instance of Data Flow

Given a process instance of P, its input data can be presented with a multi-set of IA_P, denoted as $\widehat{IA}_P$. In order to maintain the process feasibility, for artifact d in IA_P, the number of instances of d inputted, i.e., the coefficient $ms(d)$ of d in $\widehat{IA}_P$, should be equal to or greater than the number of incoming data flows transmitting d, i.e., $ms(d) \geq n$ where $InDataFlow_d = \{(d,v) \mid (d,v) \in d \times V\}$ and $n = |InDataFlow_d|$. When $ms(d) = n$, all the input instances of artifact d are consumed. When $ms(d) > n$, the actions consume n instances of d selected from $\widehat{IA}$.

Given two intermediate data flows $((u,v_1),d_1)$ and $((u,v_2),d_2)$, the propagations of artifacts between two actions can be classified into three cases:

- if $v_1 = v_2$ and $d_1 \neq d_2$, the instances of artifact d_1 and d_2 are submitted from u to v concurrently.
- if $v_1 \neq v_2$ and $d_1 = d_2$, the two instances of d_1 are submitted from u to v_1 and v_2 , respectively.
- if $v_1 \neq v_2$ and $d_1 \neq d_2$, the instance of artifact d_1 / d_2 is submitted to v_1 / v_2 .

For an activity v , if v is a consumer of artifact d , when $\nexists ((u,v),d) \in InterDataFlow(P) \rightarrow \exists (d,v) \in InDataFlow(P)$ and vice versa. For all data outflows of P , $\forall (v,d) \in OutDataFlow(P)$, d belongs to process outputs , denoted by $\widehat{OA}$, a multi-set of OA .

For the sub-process $SP2.1$ "Collect Votes", a process instance $Ins_{SP2.1}$ is generated. When $\widehat{IA}_{SP2.1} = 2Calendar$, manager accesses either calendar inputted to continue the following execution. We assume that $(EvE2.1.1, DaE2.1.2)$ $WhileLoop$ ends at the fifth iterations, such that there are five votes sent from voters. An iteration results in receiving a vote from a voter. The five iterations recurse to create a fully integrated vote tally, contributing to the process output. The set of data $\widehat{A}_{SP2.1}$ used in $Ins_{SP2.1}$ can be presented with a multi-set of $A_{SP2.1} = \{Calendar, Vote, VoteTally\}$, $\widehat{A}_{SP2.1} = 2'Calendar + 4'Vote + 1'VoteTally$.

Chapter 4. The Formulations of Well-Formed and Unstructured Control Flows

During process execution, the two issues might occur: (1) deadlock and (2) undesirable instances. The issues could be caused by ill-structured control flow, data flow or message flow. In the following subsections, we discuss these two issues of control flow. The well-formed and unstructured control flows are defined.

4.1. Well-Formed Control Flow

With typed actions and their precedence relation, various kinds of control structures can be constituted. In this thesis, the four primitive control structures, *sequential*, *parallel*, *conditional* and *iterative*, defined in [11] are concerned. These structures can be implemented by basic construction mechanism and defined within blocks. The details are listed as the followings:

1. Sequential Structure: is a sequence of actions constructed by basic construction mechanism without control nodes. For each action in the sequence, it is fired while the preceding activity is completed. The sequence is included in a sequential block.
2. Parallel Structure: is a structure implemented in *Parallel* control block. The expressive power of the block is enriched by associating with a join node which is implemented with *Discriminator*, *MultiMerge* or *NtoMJoin* mechanism.
3. Conditional Structure: is a structure implemented in *DataExclusive* and *EventExclusive* control blocks which take one of its branches to execute when upon its incoming data and event, respectively.

4. Iterative Structure: is a structure implemented in *WhileLoop* and *RepeatUntilLoop* control blocks.

An *Inclusive* control block can be implemented by a combination of *DataExclusive* and *Parallel* control blocks [11]. Similarly, the extensions of *Inclusive* control block, *InclusiveDiscriminator*, *InclusiveMultiMerge* and *InclusiveNtoMJoin*, can be represented by *DataExclusive* and *ParallelDisCriminator* / *ParallelMultiMerge* / *ParallelNtoM* control blocks also. In order to simplify our discussion, we concern merely the four primary categories of control blocks, where the blocks have no substitutions.

Within a control flow, the divergence and convergence of actions are presented by control nodes. Except control nodes, a flow diverged from an activity can be presented by a supplement arc only. Without concerning supplement arcs, a control flow is well-formed if the constraints defined in Definition 4.1 hold.

Definition 4.1. (Well-Formed Control Flow).

Given a control flow $G = (V, CF)$ of no supplement arc, i.e., $\forall (u,v) \in CF : isExtended(u,v) \neq true$, G is *well-formed* if and only if G is constructed based on the events, tasks and control blocks, defined in our control flow model, and any two control blocks within the flow can be *nested* but not *overlapped*.

When the control blocks in a well-formed control flow are represented recursively with the notation for sub-process in BPMN, the flow can be reduced to a composite action presented by sub-process. Whether the control flow leads to deadlocks and/or generate accidental instances, that will never be accessed and destroyed, is easier to indicate [29][32][33][34].

The same perspective can also be applied to a control flow including supplement arc(s), if the sub-flows connected by supplement arc(s) are well-formed. Such control flow is well-formed also. Otherwise, the process is *unstructured*. Without concerning data and message flow, every well-formed process is well-behaved [45], as Definition 4.2.

Definition 4.2. (Well-Behaved Control Flow).

Given a control flow $G=(V,CF)$, G is *well-behaved* if and only if G neither leads to deadlock nor generates undesirable instances.

4.2. Unstructured Control Flow

A control flow is unstructured when one or more restrictions for well-formed property, *pairwise restrictions* and *nesting structure*, is violated. The unstructured control flows violating the pairwise restrictions can be classified into two cases:

1. *Mismatched Structure*: a control block is bounded with a *mismatched pair* of control nodes, e.g., $ParallelSplit$ and $ExclusiveJoin$.
2. *Unpaired Structure*: a split/join node is included in a control flow without a corresponding join/split node.

In addition, an *improper nesting structure* in a process, defined in Definition 4.3, is constructed when the *one-to-one corresponding relation* of control node, is not followed.

Definition 4.3. (Improper Nesting Structure).

Given two control block $B_1=(u_1,v_1)$ and $B_2=(u_2,v_2)$ in an control flow $G=(V,CF)$, B_1 is improperly nested with B_2, if and only if the following holds:

$$u_2 \in \left(V_{u_1}^{\overline{IsSuccessor}} \cap V_{v_1}^{\overline{IsPredecessor}} \right) \wedge v_2 \notin \left(V_{u_1}^{\overline{IsSuccessor}} \cap V_{v_1}^{\overline{IsPredecessor}} \right) \text{ or}$$

$$u_2 \notin \left(V_{u_1}^{\overline{IsSuccessor}} \cap V_{v_1}^{\overline{IsPredecessor}} \right) \wedge v_2 \in \left(V_{u_1}^{\overline{IsSuccessor}} \cap V_{v_1}^{\overline{IsPredecessor}} \right)$$

In other word, $\nexists Path(u_1, v_1)$:[6] Both u_2 and v_2 are in the path.

Either mismatched control pairs or improper nesting structures may cause *behavioural anomalies* in a process execution, but not all. There are two typical behaviour anomalies concerned: *deadlocks* and *unexpected instances*.

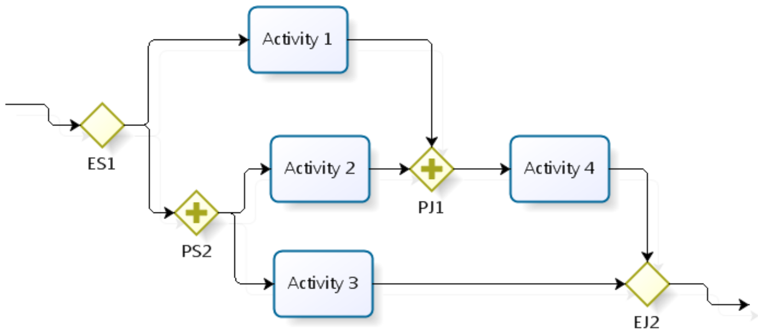


Figure 4.1 An example of overlapped structure.

Given an overlapped example, shown in Figure 4.1, to explain the two behaviour anomalies:

1. Deadlock Case: In mismatched control block $(ES1, PJ1)$, $ParallelJoin$ node $PJ1$ is deadlocked because of one or more of its incoming flows is unexecuted.
2. Unexpected Instance Case: In mismatched control block $(PS2, EJ2)$, the activities of the two branches diverged from $ParallelSplit$ node $PS2$, e.g., activity 2 and 4 or 3, are remained in workflow engine unexpectedly if another one arrives $ExclusiveJoin$ node $EJ2$ earlier.

[6] Path(u,v) denotes a path from u to v , a sequence of vertices in a control flow G=(V,CF), such that each node is connected to the next vertex in the sequence.

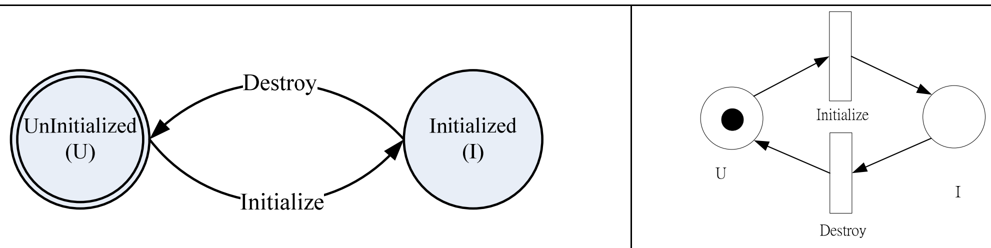
Chapter 5. The Methods for Transforming BPMN Process into H_C^T PNET

In order to analyze a business process $BP = (PP, A, M, MF, \widetilde{MF}, PF, \tilde{P})$ where $PP = \{P_i \mid i = 1..n, n \geq 1\}$, each private process P_i in BP is transformed into a H_C^T PNet $HNet_i$. All these H_C^T PNets generated are stored in set $Net = \{HNet_i \mid i = 1..n, n \geq 1\}$. The control, message and data flows of process are transformed into H_C^T PNet modules by their corresponding methods. These transformation methods are discussed in the followings.

5.1. State Transitions of Process Instance with PNet

A process instance is operated by a set of legal operations. The action executed by WfMS for manipulating a process instance executes the legal operation(s) only. Each atomic operation of a process instance can be regarded as one of the followings, regardless of its semantic meaning:

1. Initialize: an operation that instantiates a private process within a business process.
2. Destroy: an operation that deletes a process instance within a WfMS.



(a) A state transition diagram of a process instance

(b) A PNet presenting the state transitions of a process instance

Figure 5.1 Two different presentations of the state transitions of a process instance.

Figure 5.1 (a) shows the state transition diagram of a process instance. There are two possible states, “UnInitialized” and “Initialized”, of an instance as *Initialize* or *Destroy* operation occurs. “UnInitialized” state represents the initial state of a process. “Initialized” represents the state after an *Initialize* operation is performed. The state of an instance is transformed from “Initialized” to “UnInitialized” when *Destroy* operation is executed.

Figure 5.1 (b) depicts the corresponding PNet $PNet_{in} = (P, T, F, m_0)$ of the diagram shown in Figure 5.1 (a). The two places of $PNet_{in}$ present “UnInitialized” and “Initialized” states, respectively. The *Initialize* and *Destroy* operations are transformed into the *Initialize* and *Destroy* transitions. The input and output arcs of these transitions connect the places and transitions. The initial state of a process is “UnInitialized”, i.e., the initial marking M_0 of $PNet_{in}$ is $(1,0)$ while the place array is (U, I).

5.2. Transformation Method for Control Flows – $Method_{CF}$

Let private process P_i in BP be transformed into H_C^T PNet $HNet_i$. A global clock, whose cycle is z time units, is introduced in $HNet_i$. Initially, $HNet_i$ is empty. The elements in P_i are transformed to their corresponding H_C^T PNet modules one by one. The H_C^T PNet modules generated are added into $HNet_i$. The link of two different H_C^T PNet modules is denoted with dotted link. The firing interval of transition t added into $HNet_i$ is $[0, z]$ when t's corresponding activity has no time limitation. $HNet_i$ has a timed token at least, typed with two attributes $PNet_{in}$

and time, to denote P_i's execution status, i.e., the execution order of the actions in P_i is represented by a series of movements of the token(s). Such a token is called *control token* here. All the transitions in $HNet_i$ cannot be fired without the token.

5.2.1. Rules for Transforming Basic Elements

Our process model is designed based on the elements listed in Table 5.1. In this table, the element whose counterpart in the rightmost column is ◎ is a *basic* element, the element whose counterpart in the rightmost column is ○ is an *advanced* element, and the rest whose counterpart is empty are not concerned.

Most process models, e.g., [11][24][29][32][33][34][48], are designed based on the basic elements. These basic elements can be transformed into H_C^T PNet modules with Rule 1 to 7, respectively. The H_C^T PNet modules are depicted in Figure 5.2 and Figure 5.4 where the place(s) denoted with dotted line is used to link H_C^T PNet modules of two connecting BPMN actions. Such a place can be identified by a pair $p(a,b)$ where a and b are the names of two connected actions.

During the transformation, when a basic element n is reached, n can be transformed with the following rules:

Rule1. *If n represents a none start event, i.e., $n.EC = Start$ and $n.ET = None$, and n has only one direct successor y, a place denoted with p_n and an atomic transition denoted with t_n are added into $HNet_i$. A direct arc (p_n, t_n) connecting the two elements is created.*

1-1. The color domain of place p_n is $C(p_n) = \{PNet_{in}\}$ and the token elements of place p_n are $((PNet_{in}, (1,0)), @r)$ and $((PNet_{in}, (0,1)), @r)$.

- When there is a token tk with value $\left(\left(PNet_{in},(1,0)\right),@r\right)$ in p_n, a request of creating an instance of P_i is made by a participant at time r.
- When there is a token tk with value $\left(\left(PNet_{in},(0,1)\right),@r\right)$ in p_n, a process instance of P_i is created by P_i's WfMS at time r.
- When the value of tk is changed from $\left(\left(PNet_{in},(1,0)\right),@r_1\right)$ to $\left(\left(PNet_{in},(0,1)\right),@r_2\right)$, $r_1 < r_2$, the process instantiation request given at r_1 is accomplished at r_2, i.e., the participant is able to execute the actions in P_i after r_2.

1-2. The variable domain of transition t_n contains the variables typed with $PNet_{in}$ only, i.e., $Type\left(Var\left(G(t_n)\right)\right) = Type\left(Var\left((p_n,t_n)\right)\right) = \{PNet_{in}\}$.

- The guard expression $G(t_n)$ is $Var\ in == (0,1)$ and the arc expression $A(p_n,t_n)$ is $Var\ in$.
- t_n is fired immediately, when a token tk associated with value $\left(\left(PNet_{in},(1,0)\right),@r\right)$ is added into p_n.

Rule2. *If n represents a none end event, i.e., $n.EC = End$ and $n.ET = None$, and the direct predecessor of n is x, a place denoted with p_n and an atomic transition denoted with t_n are added into $HNet_i$. A direct arc (t_n,p_n) connecting the two elements is created.*

2-1. The definition of color domain of place p_n is the same as in Rule 1-1.

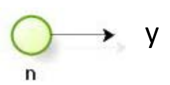
- When there is a token tk with value $\left(\left(PNet_{in},(0,1)\right),@r\right)$ in p_n, a request of terminating P_i is made by a participant at time r.

- When there is a token tk with value $((PNet_{in},(1,0)),@r)$ in p_n, the process instance is terminated by P_i's WfMS at time r.
- When the value of tk is changed from $((PNet_{in},(0,1)),@r_1)$ to $((PNet_{in},(1,0)),@r_2)$, $r_1 < r_2$, the process termination request given at r_1 is accomplished at r_2.

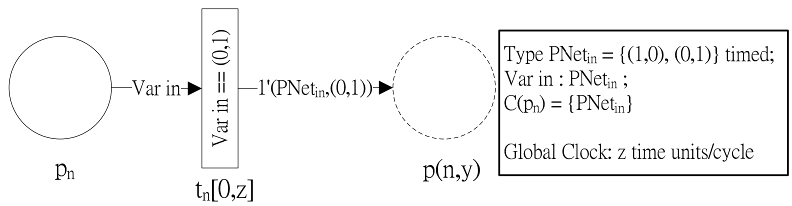
2-2. The definition of the variable domain of transition t_n is the same as in Rule1-2.

The guard and (input and output) arc expressions of the transition(s) added by applying Rules 3, 4, 5, 6 or 7 are identical to those of t_n, defined in the H_C^T PNet module of start event.

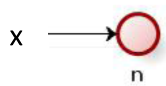
Rule3. *If n represents a task/sub-process created by the basic construction mechanism without input and output artifacts, i.e., $n.AT = Task / SubProcess$, $I_v = IA_v = \phi$ and $O_v = OA_v = \phi$, and the direct predecessor and successor of n are x and y respectively, an atomic/compound transition denoted with t_{Tn} / t_{Pn} is added into $HNet_i$. t_{Tn} / t_{Pn} has one input and output arcs.*



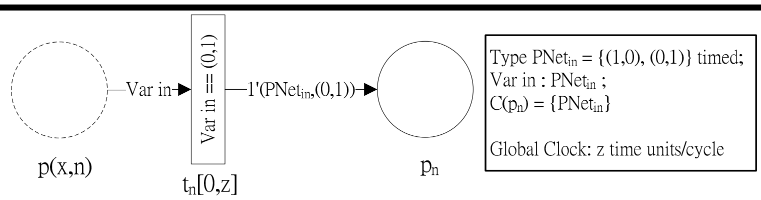
(a) A none start event.



(b) The H_C^T PNets module of none start event.



(c) A none end event.



(d) The H_C^T PNets module of none end event.

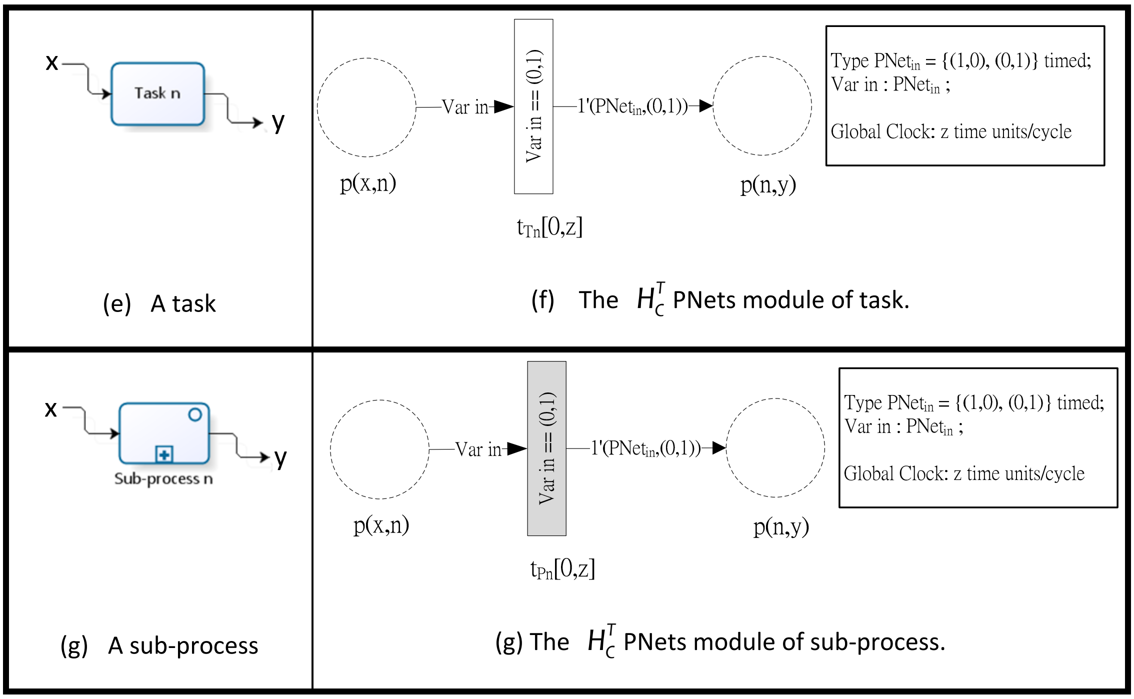


Figure 5.2 The mapping of the elements addressed in [29].

When n is a sub-process, $HNet_i$ connects the H_C^T PNet of n's expansion with two additional transition $t(x,call\ n)$ and $t(return\ n,y)$. The two transitions are used to model the invocation of sub-process n and return the control back to $HNet_i$ when n is completed. The details are shown in Figure 5.3.

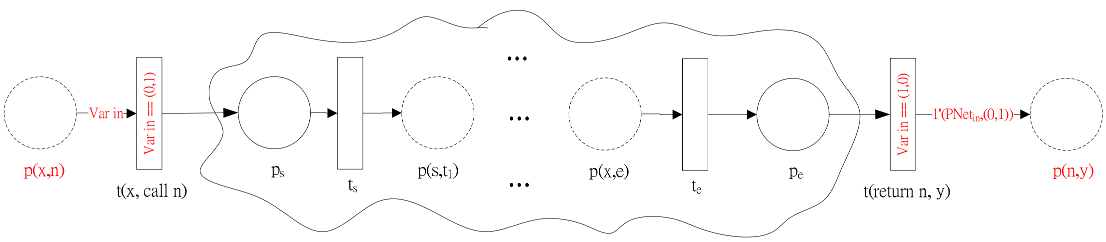


Figure 5.3 Combining the expansion of a sub-process and parent net.

Rule4. *If* n *is a data-based ExclusiveSplit control node and the direct successors of* n *are Activity* 1 *to* m, $m \geq 2$, *for each succeeding Activity* i, *an atomic transition, denoted as* $t(n,A_i), 2 \leq i \leq m$, *is added into* $HNet_i$. *Transition* $t(n,A_i)$ *has one input and output arc. The input arcs of the transitions added starts from place* $p(x,n)$.

Rule5. *If* n *is a data-based* ExclusiveJoin *control node and the direct predecessors of* n *are* Activity 1 *to* m, $m \geq 2$, *for each preceding* Activity i, *an atomic transition, denoted as* $t(n,A_i)$, $2 \leq i \leq m$, *is added into* $HNet_i$. *The number of* $t(n,A_i)$ *'s input and output arcs are one. The output arcs of the transitions added are joined at place* $p(n,y)$.

Rule6. *If* n *is a* ParallelSplit *control node and the direct successors of* n *are* Activity 1 *to* Activity m, $m \geq 2$, *an atomic transition* t_n *is added into* $HNet_i$. *Transition* t_n *has one input arc and* m *output arcs.*

Rule7. *If* n *is a* ParallelJoin *control node and the direct predecessors of* n *are* Activity 1 *to* Activity m, $m \geq 2$, *an atomic transition* t_n *is added into* $HNet_i$. The t_n *has* m *input arcs and one output arc.*

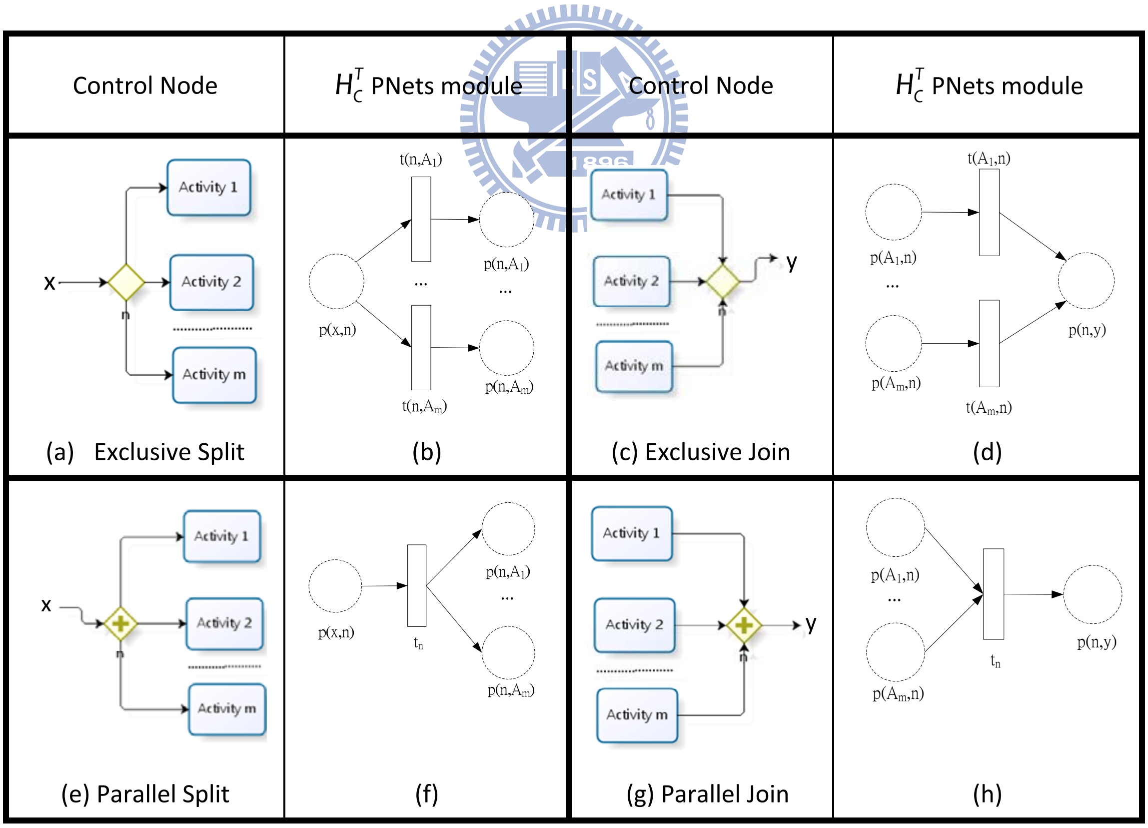


Figure 5.4 The mapping of the control nodes addressed in [29].

Table 5.1 The notations available in our process model.

BPMN				Our Process Model
Flow Objects	Activities	Task	Plain	◎
			Loop	○
			Multi-Instance	○
			Ad-Hoc	
			Compensation	
		Sub-process	Plain	◎
			Loop	○
			Multi-Instance	○
			Compensation	
	Gateways	Event-based	Exclusive	○
		Data-based	Exclusive	◎
			Inclusive	◎
			Parallel	◎
			Complex	○
	Events (Start, Intermediate, End)	Plain	Start and End	◎
		Message		○
		Timer		○
		Error		
		Cancel		
		Compensation		
		Signal		
		Multiple		
		Link		
		Terminate		
Connecting Object	Sequence Flow			◎
	Message Flow			○
	Association			
Swimlanes	Pool			○
	Lanes			
Artifacts	Data Object			◎
	Text Annotation			
	Group			
Notation: Basic elements ◎ Advanced elements ○				

5.2.2. Transformation Rules for Advanced Elements

The advanced elements can be transformed into H_C^T PNet modules with Rules 8 to 22, respectively. The rules are defined upon the sequence: (1) advanced activity and event, (2) activity involving event and (3) complex control node. In these rules, the direct predecessor and successor of the intermediate actions (activity and event) are set as x and y, respectively. The direct predecessor/successor of end/start event is set as x/y also.

(1) Advanced Activity and Event

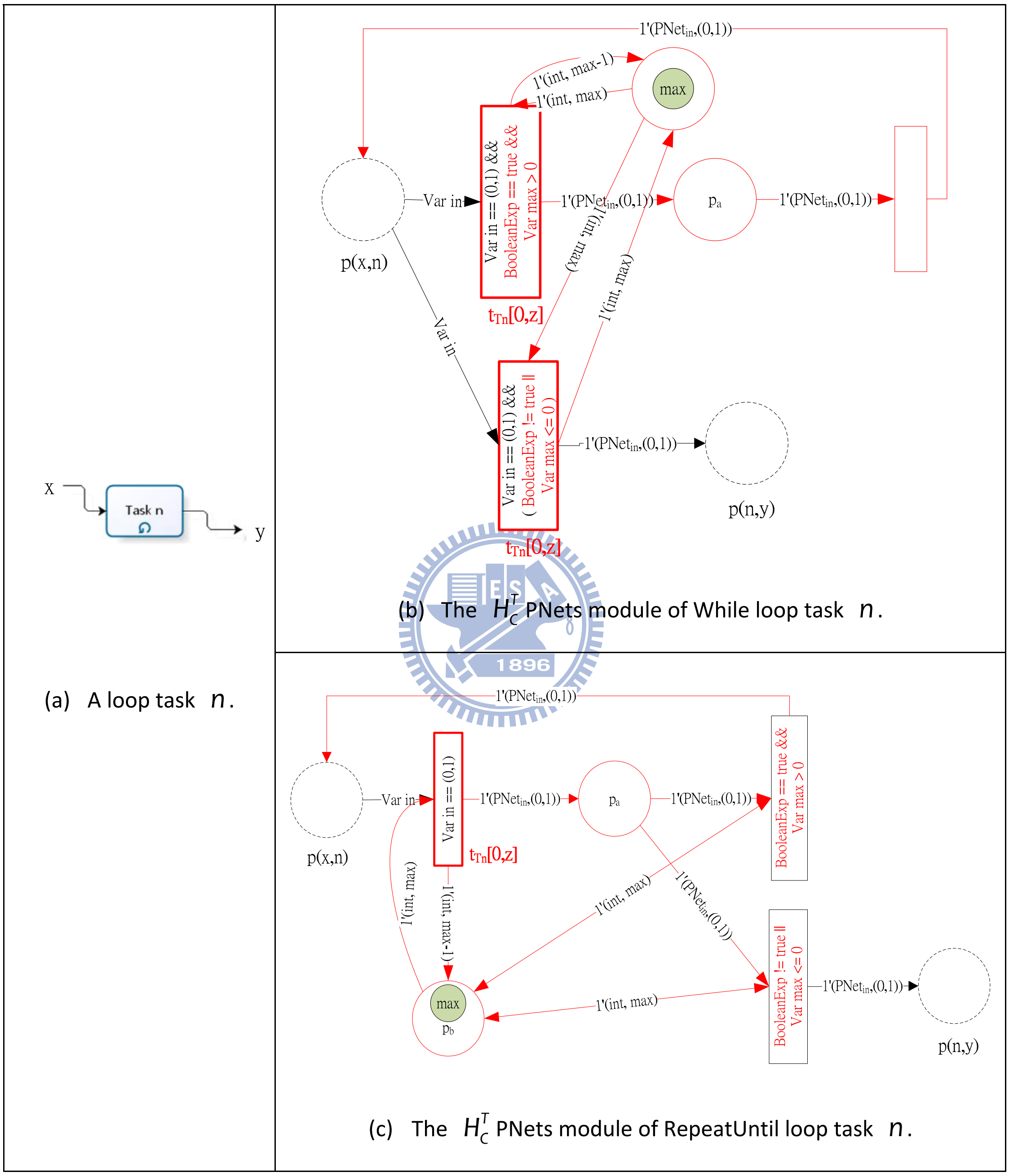
- During the transformation, when an activity (task or sub-process) n with $While/RepeatUntil$ loop structure is reached, n can be transformed with Rule 8 or 9.

Rule8. *If* n *is a loop task, i.e.,* $n.LT = Standard$, $n.EvTime = Before/After$, *and the associated evaluation expression / maximum execution times =* $BooleanExp/Maximum$, n*'s* H_C^T *PNet module is shown in* Figure 5.5 (b)/(c).

Rule9. *If* n *is a loop sub-process whose* LT, $EvTime$ *and evaluation expression / maximum execution times are the same as* Rule 8, n*'s* H_C^T *PNet module is shown in* Figure 5.5 (b)/(c) *and each atomic transition named* t_{Tn} *is replaced with a compound transition representing the sub-process.*

- During the transformation, when an activity (task or sub-process) n with multi-instance loop structure is reached, n can be transformed with Rule 10 or 11. Let the evaluation result of $NumExp$ associated with n be k, i.e., the number of instances of n is k.

Rule10. *If n is a task whose instances are performed sequentially, i.e.,* $n.LT = MultiInstance\ and\ n.Order = Sequential$, *n's* H_C^T *PNet module is shown in* Figure 5.6 (b).



(a) A loop task n.

(b) The H_C^T PNets module of While loop task n.

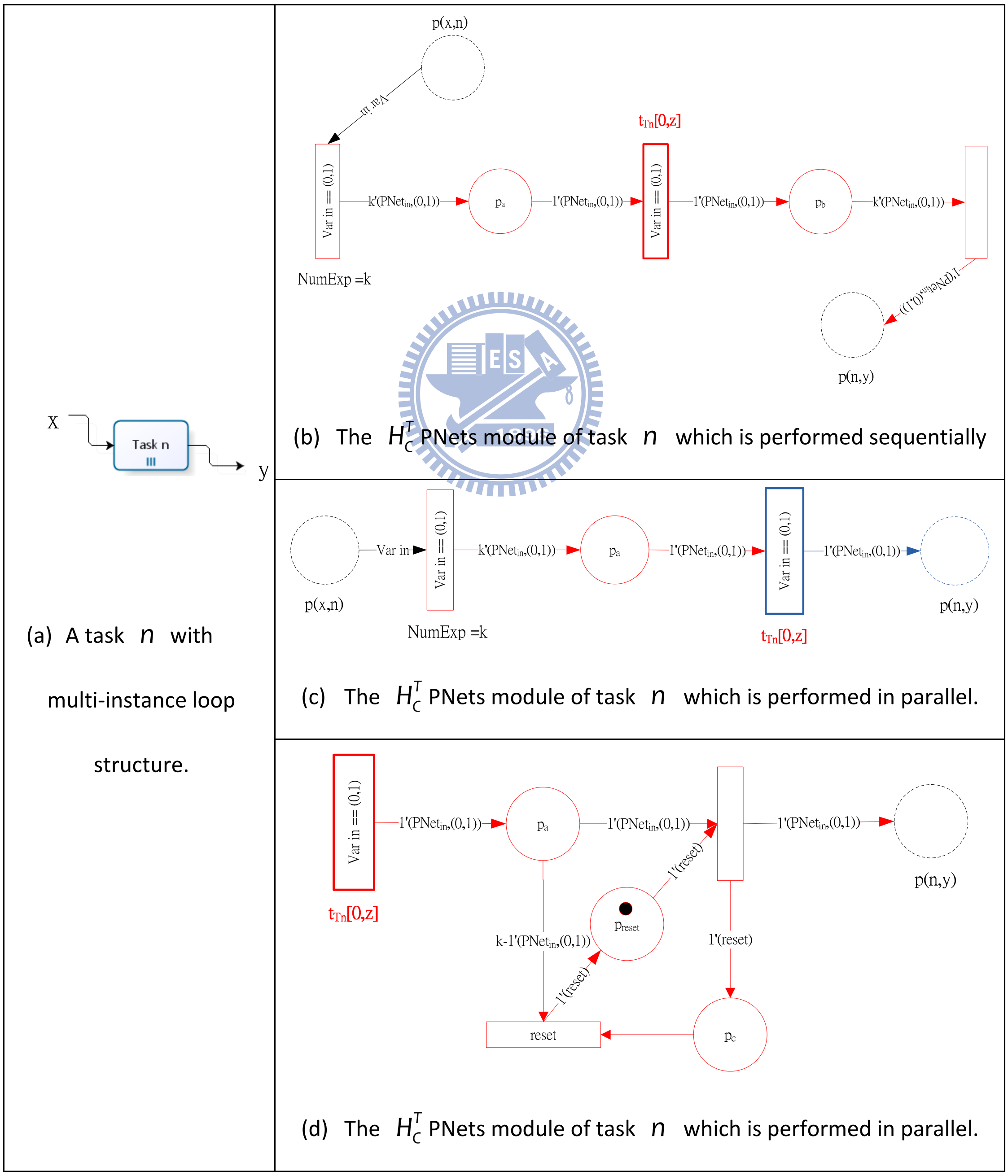
(c) The H_C^T PNets module of RepeatUntil loop task n.

Figure 5.5 Two different H_C^T PNets modules of a task with loop structure.

Rule11. *If n is a task whose instances are performed in parallel, i.e.,* $n.LT = MultiInstance\ and\ n.Order = Parallel$:

- *When* $FlowCond = None$, *n's* H_C^T *PNet module is in* Figure 5.6 (c).

- *When* $FlowCond = One$, n*'s* H_C^T *PNet module is in* Figure 5.6 (c), *but transition* $t_{Tn}[0,z]$, *place* $p(n,y)$ *and arc* $A(t_{Tn}, p(n,y))$ *are replaced with the net shown in* Figure 5.6 (d).
- *When* $FlowCond = All$, n*'s* H_C^T *PNet module is in* Figure 5.6 (c) *but transition* $t_{Tn}[0,z]$, *place* $p(n,y)$ *and arc* $A(t_{Tn}, p(n,y))$ *are replaced with the net shown in* Figure 5.6 (e).



(a) A task n with multi-instance loop structure.

(b) The H_C^T PNets module of task n which is performed sequentially

(c) The H_C^T PNets module of task n which is performed in parallel.

(d) The H_C^T PNets module of task n which is performed in parallel.

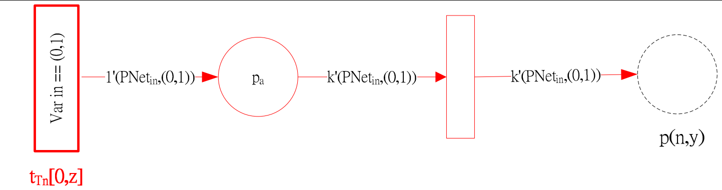
	 (e) The H_C^T PNets module of task n which is performed in parallel.

Figure 5.6 Four different H_C^T PNets modules of a task with multi-instance loop structure.

Rule12. *If n is an intermediate event, i.e., $n.EC = Intermediate$, a transition denoted with t_n is added into $HNet_i$.*

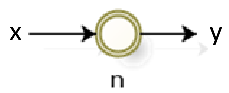
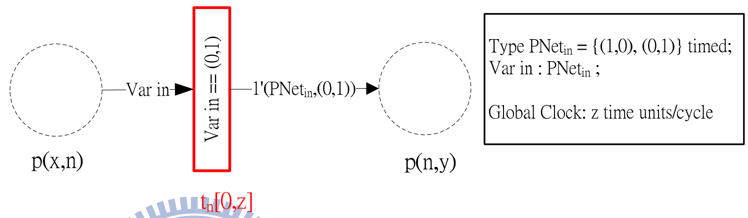
 (a) An none intermediate event.	 (b) The H_C^T PNets module of none intermediate event.

Figure 5.7 The H_C^T PNets module of intermediate event.

- During the transformation, when an event n with time limitation or message receiver/dispatcher is reached, n can be transformed with the following rules.

Rule13. *If n is a start/intermediate event and n is timed, i.e., $n.ET = Time$, and the value of n's timer attribute is $[r_1, r_2]$, $0 \le r_1 \le r_2 \le z$,* Rule1/Rule12 *is applied respectively. Then, the firing interval of t_n is changed from $[0, z]$ to $[r_1, r_2]$.*

Rule14. *If n is a start/intermediate event and n is a message receiver, i.e., $n.ET = Message$ and $n.InMessage = meg$,* Rule1/Rule12 *is applied respectively.*

Then, a place denoted with p_{meg} *is added into the net, generated by* Rule1/Rule12, *and arc* $A(p_{meg},t_n)$ *and* $A(t_n,p_{meg})$ *are created.*

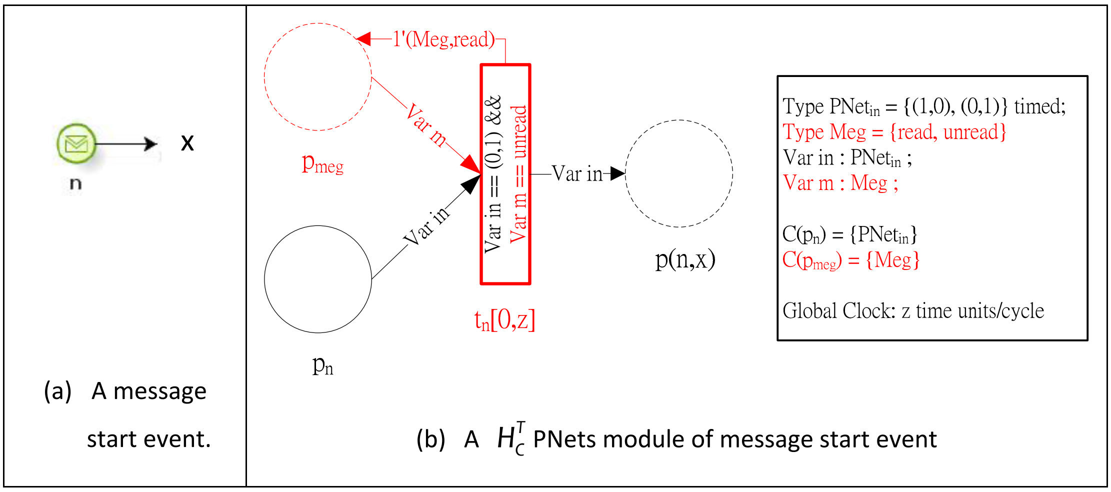
14-1. The color domain of place p_{meg} is $C(p_{meg})=\{Meg\}$ and the token elements of place p_{meg} are $(Meg,'read')$ and $(Meg,'unread')$.

- When there is a token tk with value $(Meg,unread)$ in p_{meg}, a message is sent from other participant and not consumed by the participant yet.
- When there is a token tk with value $(Meg,read)$ in p_{meg}, the message sent from other participant is consumed.

14-2. The variable domain of transition t_n contains the variables typed with $PNet_{in}$ and Meg only, i.e., $Type(Var(G(t_n)))=Type(Var((p_n,t_n)))=\{PNet_{in},Meg\}$.

- The guard expression $G(t_n)$ is $Var\ in==(0,1)\wedge Var\ m==unread$. The arc expressions of input arcs, $A(p_n,t_n)$ and $A(p_{meg},t_n)$, are $Var\ in$ and $Var\ m$, respectively. The arc expressions of output arcs, $A(t_n,p_{(n,x)})$ and $A(t_n,p_{meg})$, are $Var\ in$ and $1'(read)$, respectively.
- t_n is fired immediately, when there are two tokens with value $((PNet_{in},(1,0)),@r)$ and $(Meg,unread)$ in p_n and p_{meg}, respectively.

The H_C^T PNet module generated by applying Rule1 and Rule14 on message start event n is shown in Figure 5.8 (b).



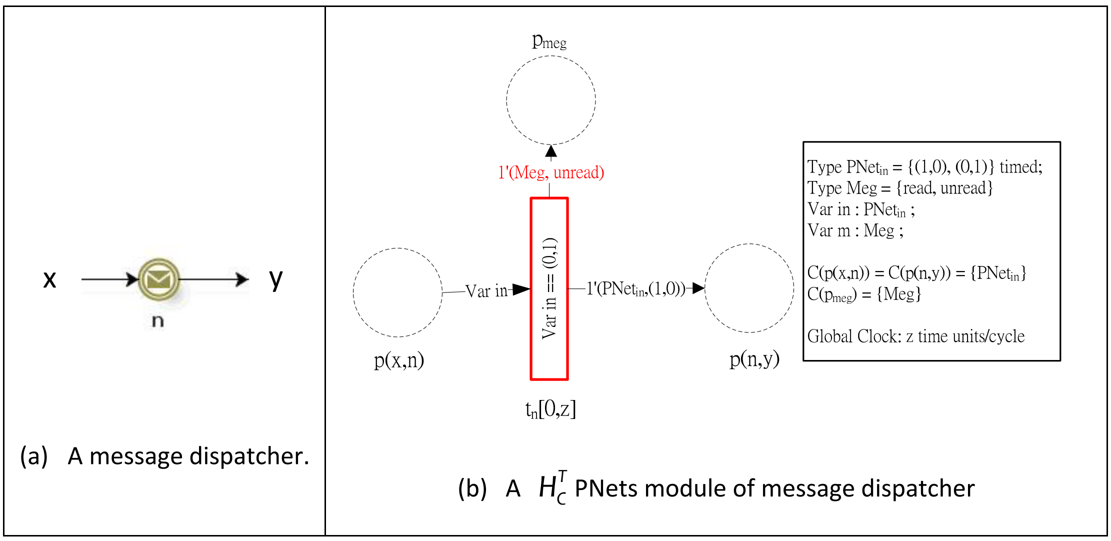
(a) A message start event.

(b) A H_C^T PNets module of message start event

Figure 5.8 Two different presentations of message start event.

Rule15. *If* n *is an intermediate/end event and* n *is a message dispatcher, i.e.,* $n.ET = Message$ *and* $n.OutMessage = meg$, Rule12/Rule2 *is applied respectively. Then, a place denoted with* p_{meg} *is added into the net generated by* Rule12/Rule2 *and the arc* $A(t_n, p_{meg})$ *is created.*

When t_n is fired, a token with value $((PNet_{in},(0,1)),@r)$ in place $p_{(x,n)}$ is removed and the tokens with value $(Meg,unread)$ and $((PNet_{in},(0,1)),@r)$ are added into p_{meg} and $p_{(n,y)}$, respectively. The H_C^T PNet module generated by applying Rule12 and Rule15 on intermediate message dispatcher n is shown in Figure 5.9 (b).



(a) A message dispatcher.

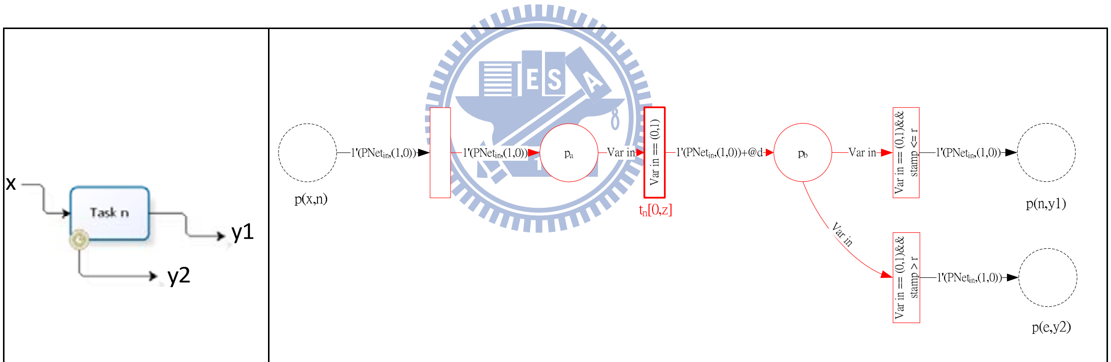
(b) A H_C^T PNets module of message dispatcher

Figure 5.9 Two different presentations of intermediate message dispatcher.

(2) Activity Involving Event

- During the transformation, when an activity (task or sub-process) n involving an event e is reached, n can be transformed by Rule 16, 17, 18, 19 or 20. Here, event e is associated with a time limitation or a message receiver/dispatcher. Let n's direct successors be y_1 and y_2. y_2 is connected by a supplement arc.

Rule16. *If* n *is a task and the value of timer attribute of* n*'s timing event* e *is* $[r_1, r_2]$, $0 \le r_1 \le r_2 \le z$, n*'s* H_C^T *PNet module is designed in* Figure 5.10 (b). *Let the time stamp associated with control token be* $stamp$.



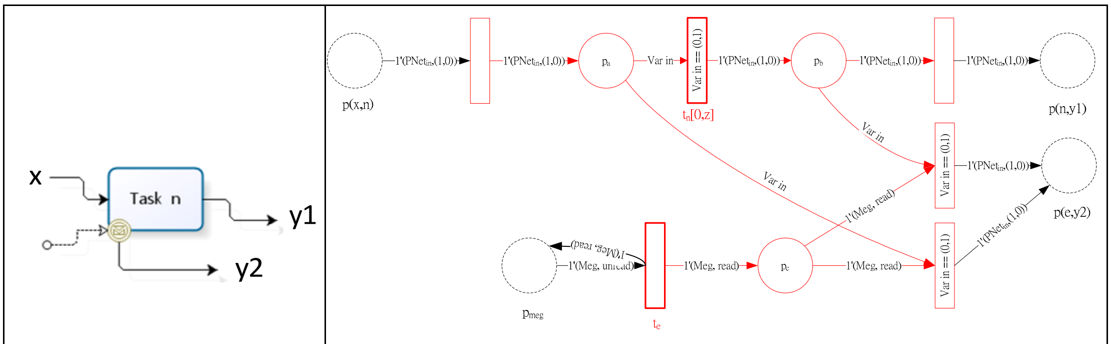
(a) Task involving a timing event.

(b) A H_C^T PNets module of a task involving a timing event.

Figure 5.10 Two different presentations of a task involving a timing event.

Rule17. *If* n *is a sub-process and the value of timer attribute of* n*'s timing event* e *is* $[r_1, r_2]$, $0 \le r_1 \le r_2 \le z$, n*'s* H_C^T *PNet module is in* Figure 5.10 (b) *while* t_n *is represented with a compound transition.*

Rule18. *If* n *is a task associated with a message receiver* e, n*'s* H_C^T *PNet module is in* Figure 5.11 (b) *where transition* t_n *is the body of* n.

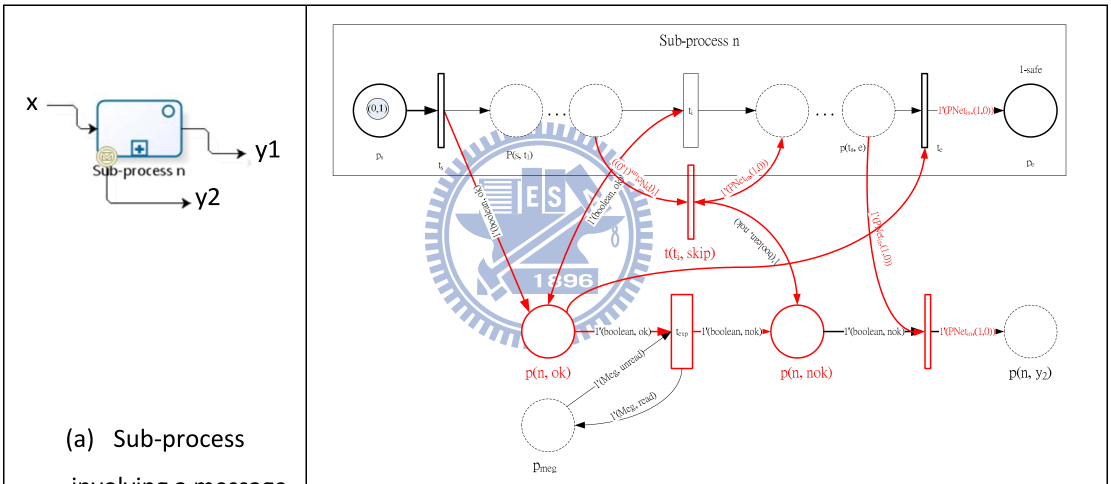


(a) Task involving a message receiver.

(b) A H_C^T PNets module of a task involving a message receiver.

Figure 5.11 Two different presentations of a task involving a message receiver.

Rule19. *If* n *is a sub-process associated with a message receiver* e*,* n*'s* H_C^T *PNet module is in* Figure 5.12 (b) *where the subnet in block is the body of* n.

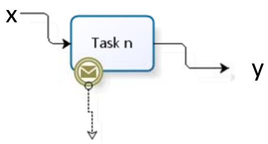


(a) Sub-process involving a message receiver.

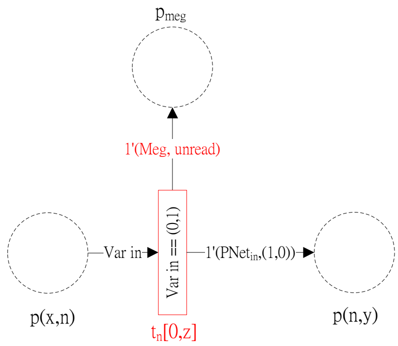
(b) A H_C^T PNets module of a sub-process involving a message receiver.

Figure 5.12 Two different presentations of a sub-process involving a message receiver.

Rule20. *If* n *is a task and associated with a message dispatcher* e*,* n*'s* H_C^T *PNet module is in* Figure 5.13 (b).



(a) Task involving a message dispatcher.



(a) A H_C^T PNets module of a task involving a message dispatcher.

Figure 5.13 Two different presentations of a task involving a message dispatcher.

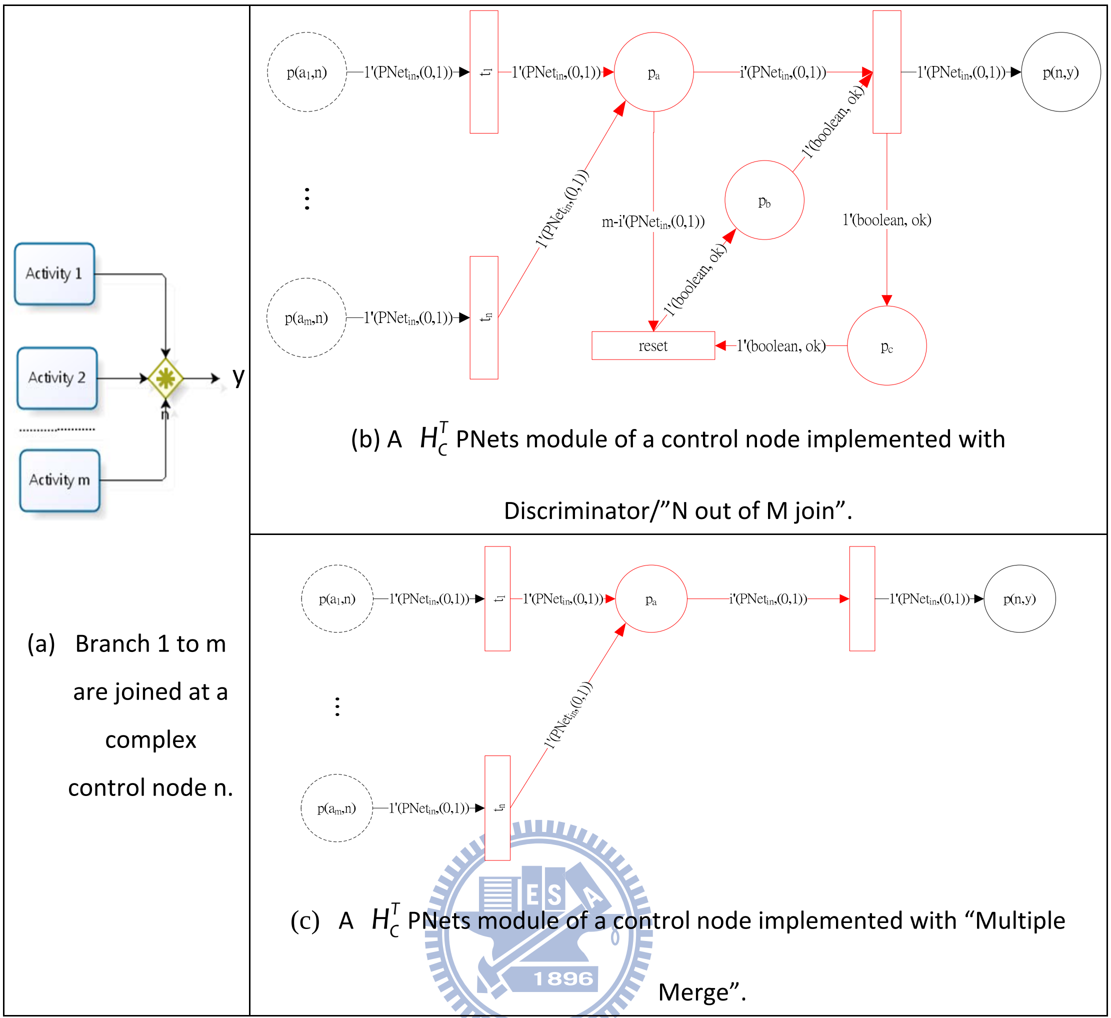
(3) Complex Control Node

- During the transformation, when a complex control node n implemented with advanced join mechanism is reached, n can be transformed by Rule 21 or 22.

Rule21. *If complex control node* n *is implemented with Discriminator or "N out M join" mechanism,* n*'s* H_C^T *PNet module is in* Figure 5.14 (b).

When n is implemented with Discriminator mechanism, variable i used in the module generated is set with 1. Otherwise, i is set with M.

Rule22. *If complex control node* n *is implemented with "Multiple Merge" mechanism,* n*'s* H_C^T *PNet module is in* Figure 5.14 (c).



(a) Branch 1 to m are joined at a complex control node n.

(b) A H_C^T PNets module of a control node implemented with Discriminator/"N out of M join".

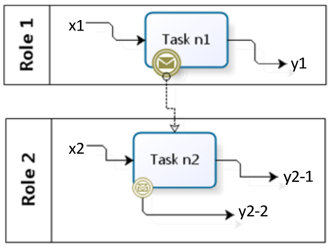
(c) A H_C^T PNets module of a control node implemented with "Multiple Merge".

Figure 5.14 Different presentations of a complex control node implemented with different mechanisms.

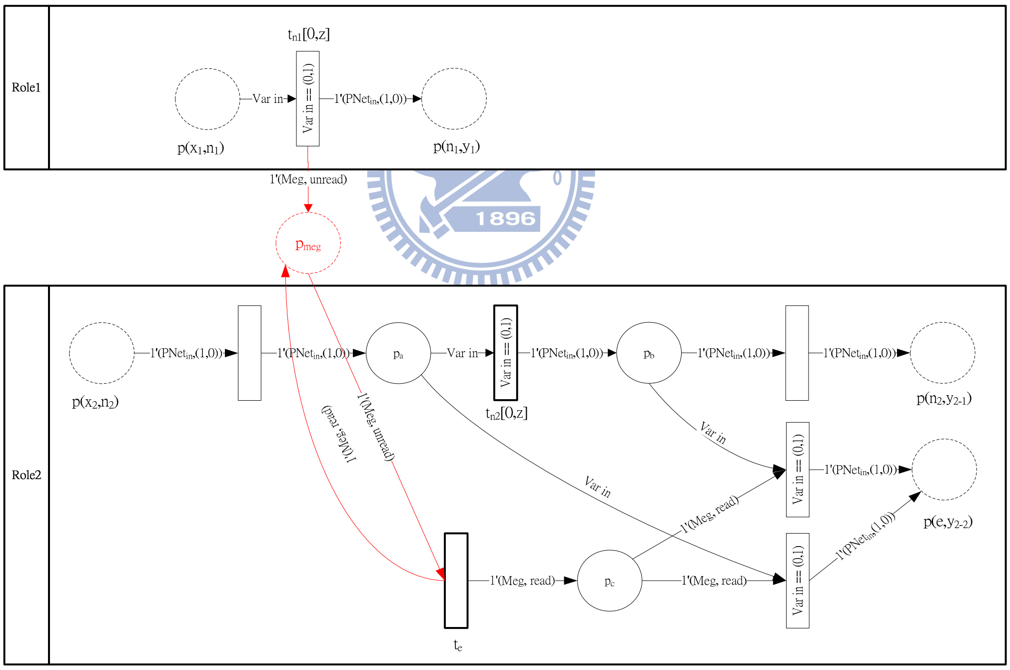
5.3. Transformation Method for Message Flows – $Method_{MF}$

A business process in BPMN may contain the following types of message flows: (1) task to task, (2) task to start event, (3) task to intermediate event , (4) intermediate event to task, (5) intermediate event to start event, (6) intermediate event to intermediate event, (7) end event to task and (8) end event to start event. These message flows can be transformed into H_C^T PNets modules with Rule23 to Rule30, respectively. In these rules, the message flows are started from action n_1 to action n_2. Each rule adopts several rules in previous section where the rules applied to the same object are executed according to the description order.

Rule23. *If message flow* (n_1,n_2) *is created between task* n_1 *involving a message dispatcher and task* n_2 *involving a message receiver,* (n_1,n_2) *'s* H_C^T *PNet module is in* Figure 5.15 (b) *created by combining the two* H_C^T *PNets modules, generated by* Rule20 *and* Rule18, *with the places denoted with* p_{meg}.



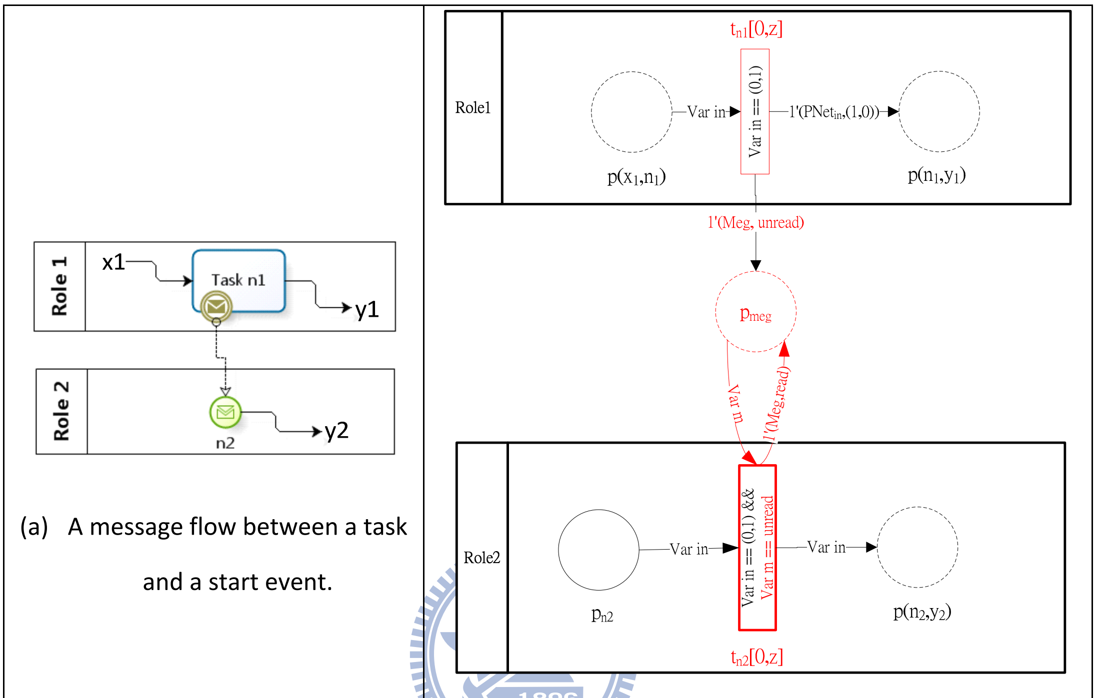
(a) A message flow between two tasks.



(b) A H_C^T PNets module of a message flow between two tasks.

Figure 5.15 Two different presentations of a message flow between two tasks.

Rule24. *If message flow* (n_1,n_2) *is created between task* n_1 *involving a message dispatcher and start event* n_2 *with a message receiver,* (n_1,n_2) *'s* H_C^T *PNet* *module is in* Figure 5.16 (b) *created by combining the two* H_C^T *PNets modules, generated by* Rule3, Rule20 *and* Rule1, Rule14, *with the places denoted with* p_{meg}.

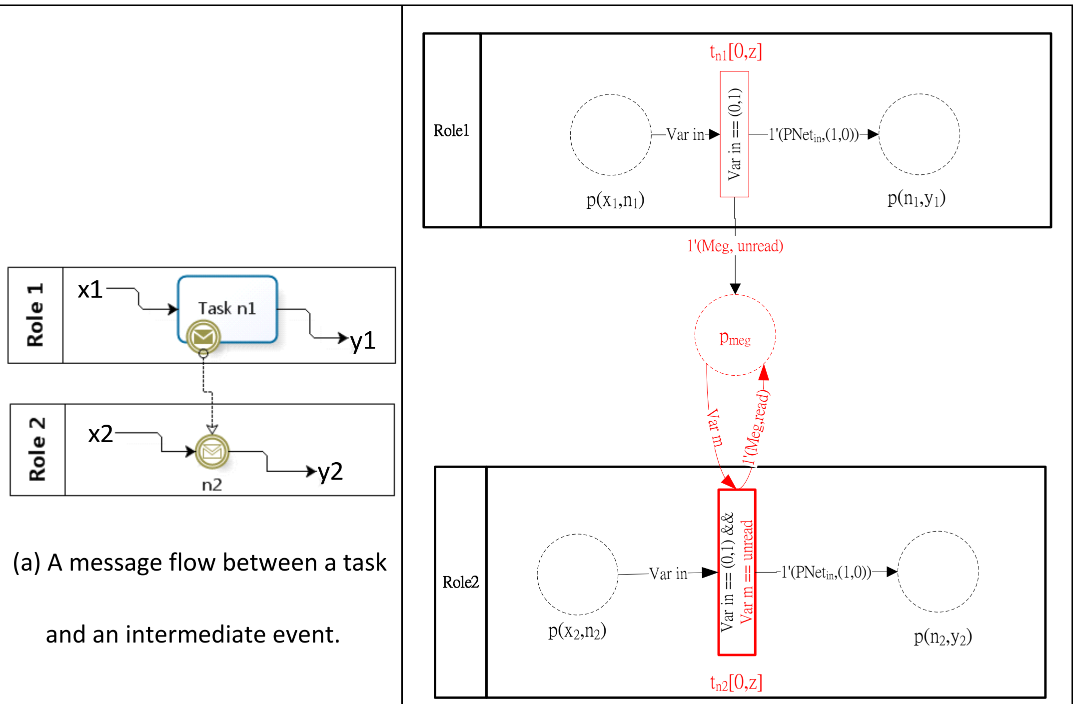


(a) A message flow between a task and a start event.

(b) A H_C^T PNets module of a message flow between a task and a start event.

Figure 5.16 Two different presentations of a message flow between a task and a start event.

Rule25. *If message flow* (n_1,n_2) *is created between task* n_1 *involving a message dispatcher and intermediate event* n_2 *with a message receiver,* (n_1,n_2) *'s* H_C^T *PNet module is in* Figure 5.17 (b) *created by combining the two* H_C^T *PNets modules, generated by* Rule20 *and* Rule14, *with the places denoted with* p_{meg}.



(a) A message flow between a task and an intermediate event.

(b) A H_C^T PNets module of a message flow between a task and an intermediate event.

Figure 5.17 Two different presentations of a message flow between a task and an intermediate event.

Rule26. *If message flow* (n_1,n_2) *is created between intermediate event* n_1 *with a message dispatcher and task* n_2 *with a message receiver,* (n_1,n_2)*'s* H_C^T *PNet module is in* Figure 5.15 (b) *created by combining the two* H_C^T *PNets modules, generated by* Rule12, Rule15 *and* Rule18, *with the places denoted with* p_{meg}.

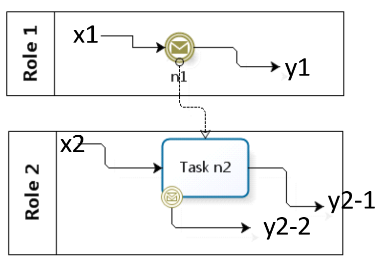


Figure 5.18 A message flow between an intermediate event and a task.

Rule27. *If message flow* (n_1,n_2) *is created between intermediate event* n_1 *with a message dispatcher and start event* n_2 *with a message receiver,* (n_1,n_2) *'s* H_C^T *PNet module is in* Figure 5.16 (b) *created by combining the two* H_C^T *PNets modules, generated by* Rule12, Rule15 *and* Rule14, *with the places denoted with* p_{meg}.

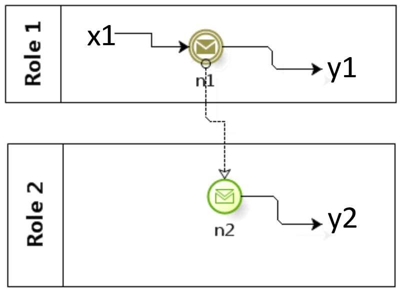


Figure 5.19 A message flow between intermediate and start events.

Rule28. *If message flow* (n_1,n_2) *is created between intermediate event* n_1 *with a message dispatcher and intermediate event* n_2 *with a message receiver,* (n_1,n_2) *'s* H_C^T *PNet module is in* Figure 5.17 (b) *created by combining the two* H_C^T *PNets modules, generated by* Rule12, Rule15 *and* Rule14, *with the places denoted with* p_{meg}.

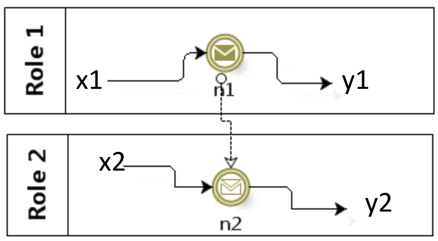
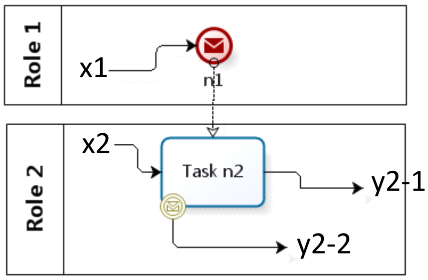
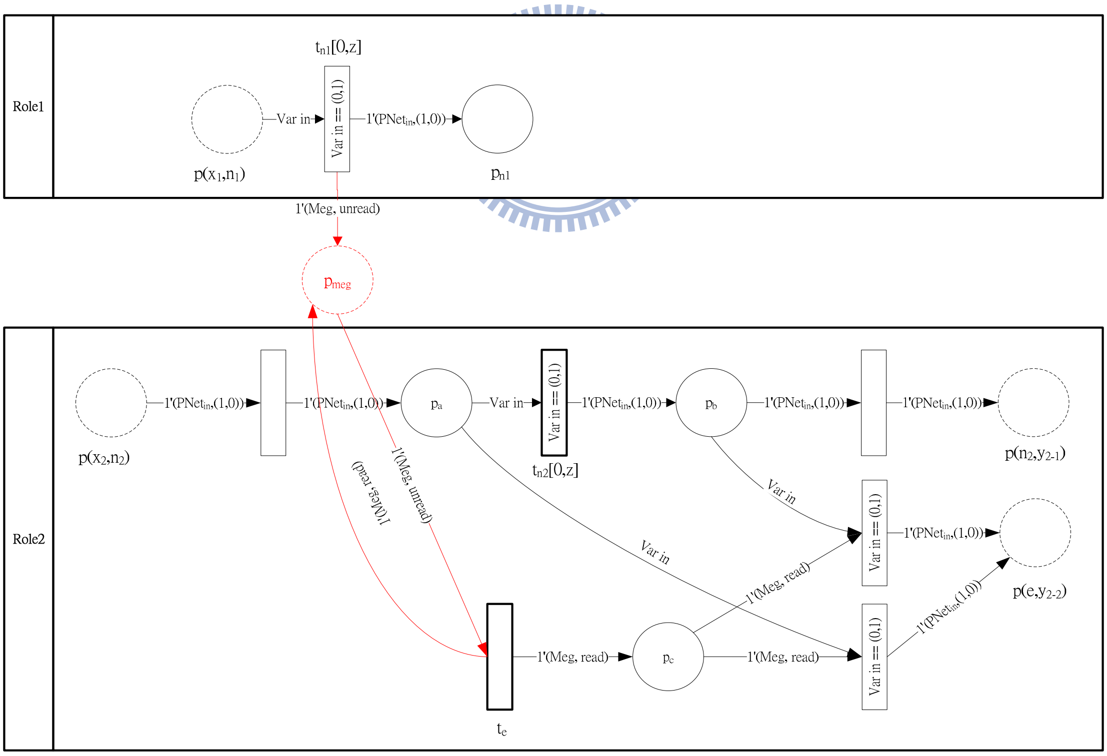


Figure 5.20 A message flow between two intermediate events.

Rule29. *If message flow* (n_1,n_2) *is created between end event* n_1 *with a message dispatcher and task* n_2 *with a message receiver,* (n_1,n_2) *'s* H_C^T *PNet module is in* Figure 5.21 (b) *created by combining the two* H_C^T *PNets modules generated by* Rule12, Rule15 *and* Rule18 *with the places denoted with* p_{meg}.



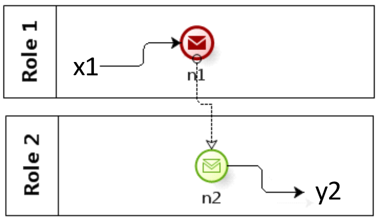
(a) A message flow between an end event and a task.



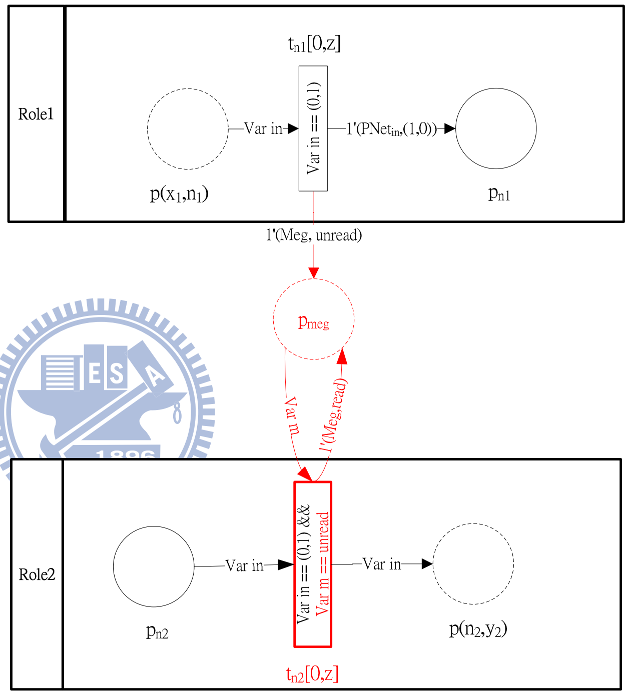
(b) A H_C^T PNets module of a message flow between an end event and a task.

Figure 5.21 Two different presentations of a message flow between an end event and a task.

Rule30. *If message flow* (n_1,n_2) *is created between end event* n_1 *with a message dispatcher and start event* n_2 *with a message receiver,* (n_1,n_2) *'s* H_C^T *PNet module is in* Figure 5.22 (b) *created by combining the two* H_C^T *PNets modules, generated by* Rule12, Rule15 *and* Rule1, Rule13, *with the places denoted with* p_{meg}.



(a) A message flow between an end event and a start event.



(b) A H_C^T PNets module of a message flow between an end event and a start event.

Figure 5.22 Two different presentations of a message flow between an end event and a start event.

5.4. Transformation Method for Data Flows – $Method_{DF}$

In a business process, the state of an artifact is transformed among the four states, “Uninitialized”, “Initialized”, “Read” and “Updated”, by the four operations, Initialize, Update, Read and Destroy. Figure 5.23 (a) shows the state transition diagram of an artifact with the four kinds of operations. The diagram can be represented with a PNet as Figure 5.23 (b). The initial state of an artifact can be represented with $(1,0,0,0)$ while the place array of the artifact PNet is (U,I,R,W). When an artifact is initialized, the state of the artifact is transformed from $(1,0,0,0)$ to $(0,1,0,0)$.

For incoming data flow (d,v), the artifact input d can be read, updated or destroyed by activity v. The three cases of incoming data flows are presented as Figure 3.17 (a), (b) and (c), respectively. The three cases can be transformed into the H_C^T PNets modules shown in Figure 5.24 when the arc expression of arc (t_v, p_d) is set with $1'(0,0,1,0)$, $1'(0,0,0,1)$ and $1'(1,0,0,0)$, respectively. After transition t_v is fired, the value of a token representing artifact d is changed to the assigned value described on the arc expression and the token is added into place P_d.

Rule31. *If* v is a reader/updater/destroyer of artifact d, data flow (d,v) *’s* H_C^T *PNet module is in* Figure 5.24 *and* $A(t_v,p_d)=1'(0,0,1,0)$ / $A(t_v,p_d)=1'(0,0,0,1)$ / $A(t_v,p_d)=1'(1,0,0,0)$.

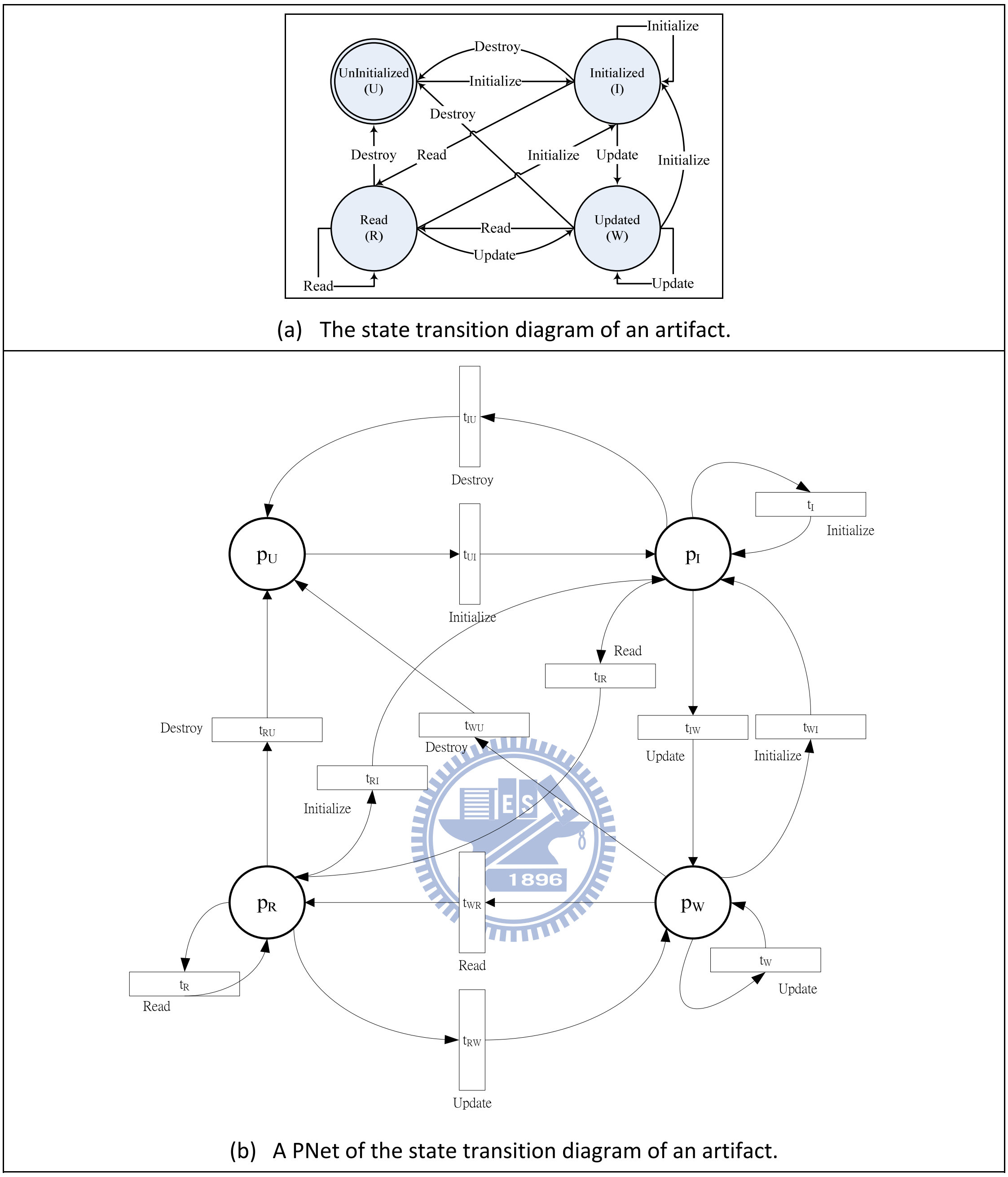


Figure 5.23 Two different presentations of the state transition of an artifact.

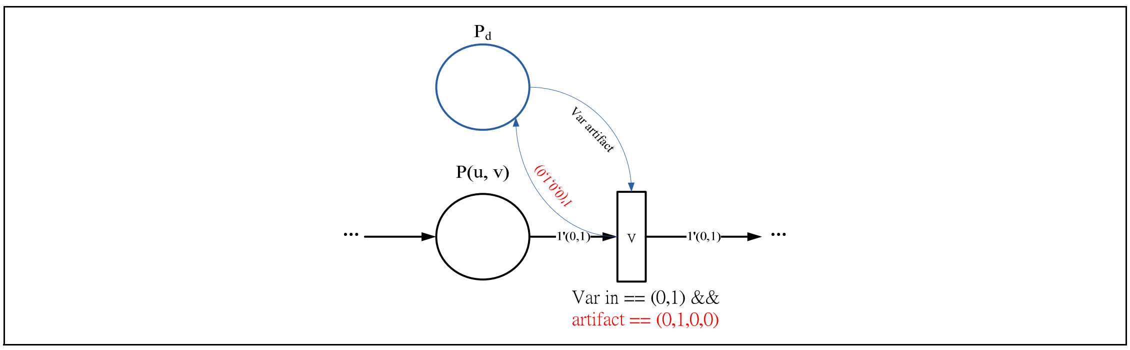


Figure 5.24 A H_C^T PNets module of incoming data flows.

The transformation rules for the intermediate data flow $((u,v),d)$ discussed in Section 3.3.3 are shown in the followings.

- When the artifact d is produced by action u and consumed by action v, place p_d for d and arc (t_u,p_d), (p_d,t_v) and (t_v,p_d) for presenting the interaction of control and data flow are added into the H_C^T PNets modules.

Rule32. *If* v is a reader of d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.25 *where* $A(t_u,p_d)=1'(0,1,0,0)$, $A(p_d,t_v)=Var\ artifact$, $A(t_v,p_d)=1'(0,0,1,0)$ and $G(t_v)=Var\ in==(0,1)\&\&Var\ \ artifact==(0,1,0,0)$.

Rule33. *If* v is a destroyer of d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.25 *where* $A(p_d,t_v)=Var\ \ artifact$, $A(t_v,p_d)=1'(1,0,0,0)$ and $G(t_v)=Var\ in==(0,1)\&\&Var\ \ artifact==(0,1,0,0)$.

Rule34. *If* v is an updater of d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.25 *where* $A(t_u,p_d)=1'(0,1,0,0)$, $A(p_d,t_v)=Var\ artifact$, $A(t_v,p_d)=1'(0,0,0,1)$ and $G(t_v)=Var\ in==(0,1)\&\&Var\ \ artifact==(0,1,0,0)$.

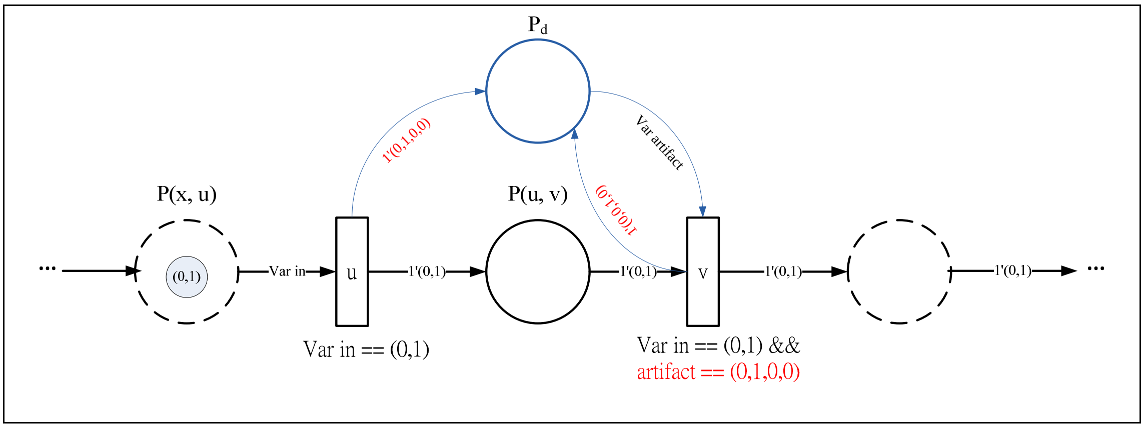


Figure 5.25 A H_C^T PNets module of intermediate data flows.

- When both action u and v are consumers of the artifact d, place p_d and arc (t_u,p_d), (p_d,t_u), (p_d,t_v) and (t_v,p_d) are added into the H_C^T PNets modules.

Rule35. *If both* u *and* v *are readers of* d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.26 *where* $A(t_u,p_d)=1'(0,0,1,0)$, $A(p_d,t_v)=Var\ \ artifact$, $A(t_v,p_d)=1'(0,0,1,0)$ *and*

$$G(t_u)=G(t_v)=Var\ in==(0,1)\&\&Var\ \ artifact!=(1,0,0,0).$$

Rule36. *If* u *is a reader and* v *is a destroyer of* d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.26 *where* $A(p_d,t_u)=Var\ \ artifact$, $A(t_u,p_d)=1'(0,0,1,0)$, $A(p_d,t_v)=Var\ \ artifact$, $A(t_v,p_d)=1'(1,0,0,0)$ *and* $G(t_u)=G(t_v)=Var\ in==(0,1)\&\&Var\ artifact!=(1,0,0,0)$.

Rule37. *If* u *is a reader and* v *is an updater of* d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.26 *where* $A(p_d,t_u)=Var\ \ artifact$, $A(t_u,p_d)=1'(0,0,1,0)$, $A(p_d,t_v)=Var\ \ artifact$, $A(t_v,p_d)=1'(0,0,0,1)$ *and* $G(t_u)=G(t_v)=Var\ in==(0,1)\&\&Var\ artifact!=(1,0,0,0)$.

Rule38. *If* u *is a destroyer and* v *is a reader of* d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.26 *where* $A(p_d,t_u)=Var\ \ artifact$, $A(t_u,p_d)=1'(1,0,0,0)$,

$A(p_d,t_v)=Var\ artifact$, $A(t_v,p_d)=1'(0,0,1,0)$ and $G(t_u)=G(t_v)=Var\ in==(0,1)\&\&Var\ artifact!=(1,0,0,0)$.

Rule39. *If* u *is a destroyer and* v *is a destroyer of* d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.26 *where* $A(p_d,t_u)=Var\ artifact$, $A(t_u,p_d)=1'(1,0,0,0)$, $A(p_d,t_v)=Var\ artifact$, $A(t_v,p_d)=1'(1,0,0,0)$ and $G(t_u)=G(t_v)=Var\ in==(0,1)\&\&Var\ artifact!=(1,0,0,0)$.

Rule40. *If* u *is a destroyer and* v *is a updater of* d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.26 *where* $A(p_d,t_u)=Var\ artifact$, $A(t_u,p_d)=1'(1,0,0,0)$, $A(p_d,t_v)=Var\ artifact$, $A(t_v,p_d)=1'(0,0,0,1)$ and $G(t_u)=G(t_v)=Var\ in==(0,1)\&\&Var\ artifact!=(1,0,0,0)$.

Rule41. *If* u *is a updater and* v *is a reader of* d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.26 *where* $A(p_d,t_u)=Var\ artifact$, $A(t_u,p_d)=1'(0,0,0,1)$, $A(p_d,t_v)=Var\ artifact$, $A(t_v,p_d)=1'(0,0,1,0)$ and $G(t_u)=G(t_v)=Var\ in==(0,1)\&\&Var\ artifact!=(1,0,0,0)$.

Rule42. *If* u *is a updater and* v *is a destroyer of* d, $((u,v),d)$*'s* H_C^T *PNet module is in* Figure 5.26 *where* $A(p_d,t_u)=Var\ artifact$, $A(t_u,p_d)=1'(0,0,0,1)$, $A(p_d,t_v)=Var\ artifact$, $A(t_v,p_d)=1'(1,0,0,0)$ and $G(t_u)=G(t_v)=Var\ in==(0,1)\&\&Var\ artifact!=(1,0,0,0)$.

Rule43. *If* u *is a updater and* v *is an updater of* d, $((u,v),d)$ *'s* H_C^T *PNet module is in* Figure 5.26 *where* $A(p_d,t_u)=Var\ artifact$, $A(t_u,p_d)=1'(0,0,0,1)$, $A(p_d,t_v)=Var\ artifact$, $A(t_v,p_d)=1'(0,0,0,1)$ *and* $G(t_u)=G(t_v)=Var\ in==(0,1)\&\&Var\ artifact!=(1,0,0,0)$.

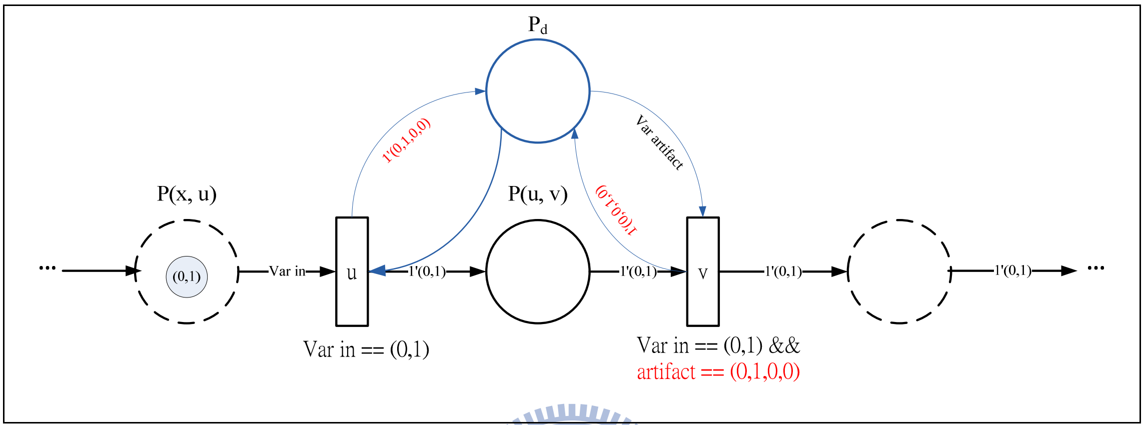


Figure 5.26 A H_C^T PNets module of intermediate data flows.

5.5. Process Transformation

Let a business process $BP=(PP,A,M,MF,\widetilde{MF},PF,\tilde{P})$ be transformed into a H_C^T PNet $Net=(TNet,TrSet,TkSet,TrFun,TkFun)$. Each kind of artifacts/messages in $A\ /\ M$ is designed with a PNet. BP is composed of private processes, P_1, P_2,..., P_n, $n\geq 1$. For private process P_i, $1\leq i\leq n$, the control flow of P_i is $ControlFlow(P_i)=(G_i,\tilde{V}_i,A_i,M_i,I_i,O_i)$ where $G_i=(V_i,CF_i)$ started from any start event in $StartSet_i$ and ended at any end event in $EndSet_i$. The data flows of P_i are in $DataFlow(P_i)$.

The transformation is designed to convert the private processes in a business process one by one. An empty H_C^T PNet is declared for the business process in the beginning. During the transformation, a sub- H_C^T PNet is created for each private process visited. The transformation of private process can be divided into two steps:

(1) Firstly, the rules defined in $Method_{CF}$ are applied to the actions visited with Breadth-first search [49]. The H_C^T PNet modules generated are appended to the sub- H_C^T PNet sequentially.

(2) Then, the rules defined in $Method_{DF}$ are applied to the data flows to generate the corresponding modules which are appended to the sub- H_C^T PNet generated in the first step.

Such a recursive operation continues until all private processes are processed. Then, the message flows between each pair of private processes are transformed by merging the corresponding sub- H_C^T PNets upon the rules defined in $Method_{MF}$. The transformation completes when all the sub- H_C^T PNets are merged. The details of transforming a business process are shown in PseudoCode1.

```
PseudoCode1 TransformBusinessProcess(PP) {
// Input: PP : a set of private processes
// Output: resultNet: a hierarchical Timed Coloured Petri Net
        Stack currentNetStack = new Stack();
        For each private process p in PP {
            currentNet = TransformControlFlow(G , StartSet);
            // G is p's control flow and StartSet is a set of p's start events

            currentNet = TransformDataFlow(currentNet, DataFlow);
            // DataFlow is a set of data flows of p

            currentNetStack.add(currentNet);
        }

        currentNet = currentNetStack .pop;
        For each net net1 in currentNetStack {
            currentNet = TransformMessageFlow(currentNet , net1);
        }
```

```
        resultNet = currentNet;
        Return resultNet;
}
```

```
PseudoCode2 TransformControlFlow(G , StartSet) {
// Input: G=(V,CF) : a directed connected graph
//         StartSet: the traverse is started from start events in StartSet.
// Output: resultNet: a hierarchical Timed Coloured Petri Net

        FIFO queue = new FIFO();

        For each vertex v in V - StartSet {
            status[v] = 'waiting';
            level[v] = null;
            parent[v] = null;
        }

        For each vertex s in StartSet { // all start events are initialized;
            status[s] = 'operating';
            level[s] = 0;
            parent[s] = null;
            queue.add(s);
        }

        while (queue != null) {
            currentVertex= queue.first;
            subNet = Method_CF(currentVertex);
            currentNet.append(subNet);
            // subNet is appended to currentNet with links, the places denoted with dotted
            // line

            For each edge (currentVertex, u) in CF {
                If (u.status == 'waiting') {
                    status[u] = 'operating';
                    level[u] = level[currentVertex] + 1;
                    parent[u] = currentVertex;
                    queue.add(u);
                }
            }
            status[currentVertex] = 'done';
        }
        resultNet = currentNet;
        Return resultNet;
}
```

```
PseudoCode3 TransformDataFlow(net , dataFlow) {
// Input: net : a result net of TransformControlFlow(G , s) of private process P
//         dataFlow : a set of data flows of P
// Output: resultNet: a hierarchical Timed Coloured Petri Net

        currentNet = net;
```

```
        For each df in dataFlow {
            subNet = Method_DF(df);
            currentNet.append(subNet);
        }
        resultNet = currentNet;
        Return resultNet;
}
```

PseudoCode4 TransformMessageFlow(net1 , net2) {

```
// Input: net1 and net2 : the results of TransformDataFlow(G1 , s1) and (G2 , s2)
//        V1 and V2: the sets of vertices of Net1 and Net2.
// Output: resultNet: a hierarchical Timed Coloured Petri Net

        currentNet = net1 + net2;
        For each vertex u in V1 {
            For each vertex v in V2 {
                If ( u == v)    currentNet .merge(u, v);
            }
        }
        resultNet = currentNet;
        Return resultNet;
}
```

Chapter 6. A Case Study

To demonstrate the methods, $Method_{CF}$, $Method_{MF}$ and $Method_{DF}$, proposed in Chapter 5, the process BP_{vote} of resolving issues through e-mail votes introduced in section 3.2.4 and 3.3.4 is adopted as an example in this section. Process BP_{vote} is composed of three private processes, $P_{workingGroup}$, $P_{manager}$ and P_{voter}. BP_{vote} has turn cycle of a week. The methods presented are applied on this example to illustrate the steps to generate the corresponding H_C^T PNets. The control, message and data flows of the example are shown in Figure 3.14 and Figure 3.19, respectively. The artifacts are stated with details in Table 3.2. The artifact usages of actions are listed in Table 3.3.

Figure 6.1 (b) shows the H_C^T PNet of private process $P_{workingGroup}$ and $P_{manager}$, shown in Figure 6.1 (a), which is generated according to the action taken order of the two processes by the three transformation methods. Because process BP_{vote} is executed weekly, in our design, a global clock counting with hours is introduced into the H_C^T PNet and the clock is reset weekly. An execution of either task $T1.1$ or $T2.2$ takes 24 hours. There is no specific execution limitation for the four tasks shown in Figure 6.1 (a).

We assume that process BP_{vote} is started to execute at 9 am on Monday. The initial marking of the H_C^T PNet is shown in the first column in Table 6.1. Let the firing sequence is

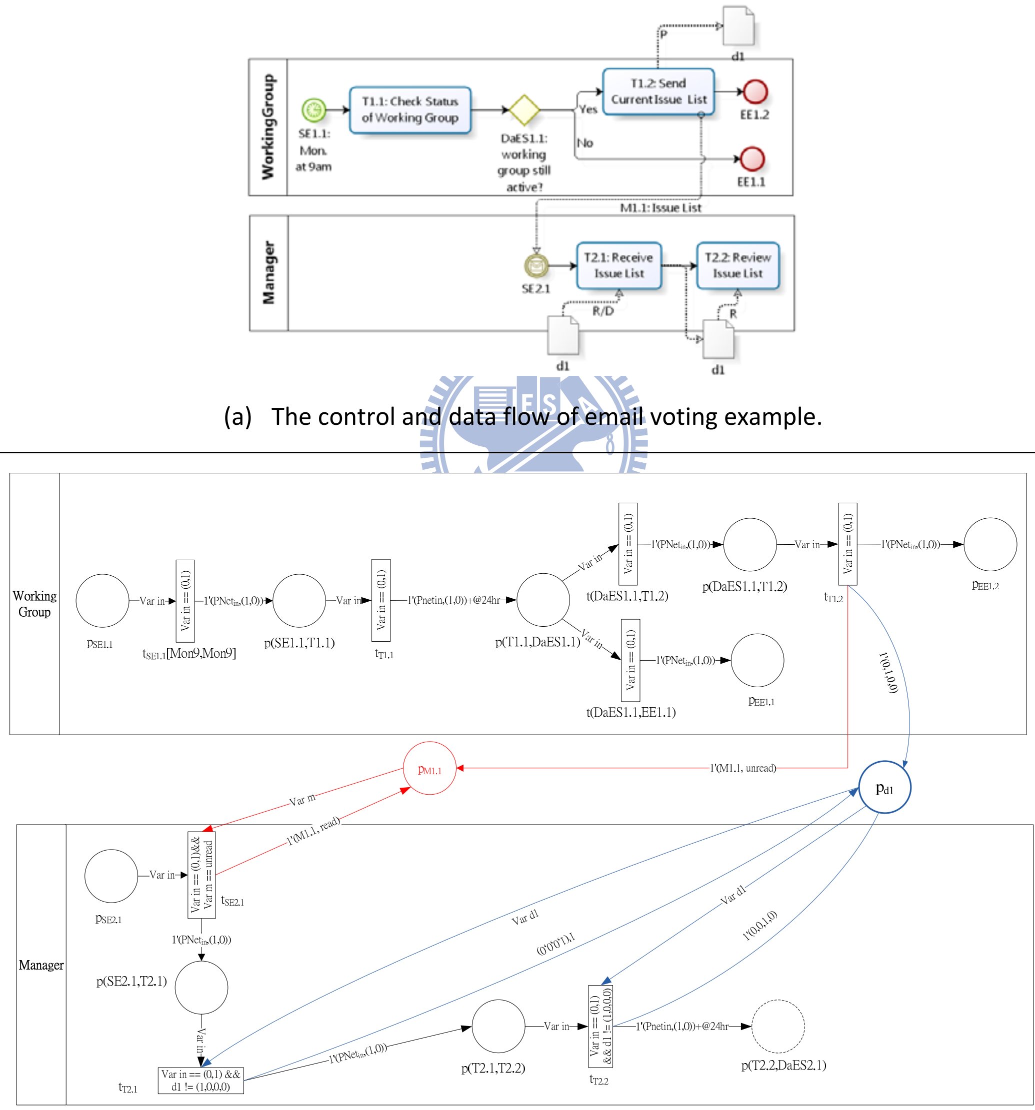
$$M_0\left[t_{sys} \succ M_1\left[t_{SE1.1} \succ M_2\left[t_{T1.1} \succ M_3\left[t_{(DaES1.1,T1.2)} \succ M_4\left[t_{T1.2} \succ M_5 \right.\right.\right.\right.\right. \text{ and}$$

$$M_6\left[t_{sys} \succ M_7\left[t_{SE2.1} \succ M_8\left[t_{T2.1} \succ M_9 \right.\right.\right. .$$

Initial marking M_0		M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9
$p_{SE1.1}$	1'(1,0)@9hr	1'(0,1)@9hr	0	0	0	0	0	0	0	0
$p(SE1.1,T1.1)$	0	0	1'(0,1)@9hr	0	0	0	0	0	0	0
$p(T1.1,DaSE1.1)$	0	0	0	1'(0,1)@33hr (Tues. 9am)	0	0	0	0	0	0
$p(DaSE1.1,T1.2)$	0	0	0	0	1'(0,1)@33hr	0	0	0	0	0
$p_{EE1.2}$	0	0	0	0	0	1'(0,1)@33hr	1'(0,1)@33hr	1'(0,1)@33hr	1'(0,1)@33hr	1'(0,1)@33hr
$p_{EE1.1}$	0	0	0	0	0	0	0	0	0	0
$p_{SE2.1}$	0	0	0	0	0	0	1'(1,0)@34hr	1'(0,1)@34hr	0	0
$p(SE2.1,T2.1)$	0	0	0	0	0	0	0	0	1'(0,1)@34hr	0
$p(T2.1,T2.2)$	0	0	0	0	0	0	0	0	0	1'(0,1)@34hr
$p(T2.2,DaSE2.1)$	0	0	0	0	0	0	0	0	0	0
$p_{M1.1}$	0	0	0	0	0	1'(0,1)@33hr (meg, unread)	1'(0,1)@33hr	1'(0,1)@33hr	1'(1,0)@34hr	1'(1,0)@34hr
p_{d1}	0	0	0	0	0	1'(0,1,0,0)@33hr (d1, uninitialized)	1'(0,1,0,0)@33hr	1'(0,1,0,0)@33hr	1'(0,1,0,0)@33hr	1'(1,0,0,0)@34hr

Table 6.1 The firing sequence of process BP_{vote}

In marking M_9, the value of artifact d_1 is transformed from $(0,0,1,0)$ to $(1,0,0,0)$ by firing transition $t_{T2.1}$, i.e., artifact d_1 is destroyed. The direct succeeding task $T_{2.2}$ cannot the artifact. In other word, transition $t_{T2.2}$ is unable to be fired because the evaluation result of $t_{T2.2}$'s guard expression is false. A deadlock happens. A missing production anomaly caused by early destruction, defined in our previous work [11], is detected.



(a) The control and data flow of email voting example.

(b) The H_C^T PNet of the example shown in (a).

Figure 6.1 Two presentations of the email voting example.

Chapter 7. Comparisons

7.1. Comparison of BPMN-based Process Models

A formal process model proposed in this paper is based on the control, message and data flows defined in BPMN. In the model, each notation for BPMN can be referred to one in [24] and [29]. The notation mappings between ours and [24] and [29] are shown in Table 7.1, Table 7.2 and Table 7.3, respectively.

Table 7.1 The mappings of the elements in message flow addressed.

Message Flow		**Our process model**	Remco et al. [24]	Y.D. Lin et al. [29]
Role	Participant and Flow Engine	**Supported**	N/A	N/A
Task to Task		**Supported**	Supported	N/A
Task to Start Event		**Supported**	Supported	N/A
Task to Intermediate Event		**Supported**	N/A	N/A
Intermediate Event to Task		**Supported**	N/A	N/A
Intermediate Event to Start Event		**Supported**	N/A	N/A
Intermediate Event to Intermediate Event		**Supported**	N/A	N/A
End Event to Task		**Supported**	Supported	N/A
End Event to Start Event		**Supported**	Supported	N/A

Table 7.2 The mappings of the elements of control flow addressed.

Control flow			**Our process model**	Remco et al. [24]	Y.D. Lin et al. [29]
Event	Timing/Message Event	Start	**Supported**	Partially Supported	N/A
		Intermediate	**Supported**	Partially Supported	N/A
		End	**Supported**	Partially Supported	N/A
Activity	Task		**Supported**	Supported	Supported
	Sub-Process		**Supported**	Supported	Supported
	Task/ Sub-Process	Activity Involving Event	**Supported**	Supported	N/A
		Standard Loop Activity	**Supported**	Supported	N/A
		Multi-Instance Loop Activity	**Supported**	N/A	N/A
Control Node	Data-Based (Well-Formed)	Exclusive	**Supported**	Supported	Supported
		Inclusive	**Supported**	Supported	Supported
		Parallel	**Supported**	Supported	Supported
		Complex	**Supported**	N/A	N/A
		Iterative	**Supported**	Supported	Supported
	Event-Based	Exclusive	**Supported**	N/A	N/A
Unstructured	Mismatched Structure		**Supported**	N/A	N/A
	Unpaired Structure		**Supported**	N/A	N/A
	Improper Nesting Structure		**Supported**	N/A	N/A

There are many ways for the artifacts to be defined and utilised in process. In BPMN, the visibility and usability of an artifact is determined by the scope of process or task. In our process model, the artifact(s) associated with a process or task is defined as the ‘input’ and ‘output’ attribute(s) of the latter. It is easier to use data channels, distinct from control channels, to analyze the artifact interactions. An artifact of multiple instances is partially supported: Our process model does not support assigning specific artifact instances to different task instances.

Table 7.3 The mapping of the elements of data flow addressed.

<table>
<tr><th colspan="2">Data flow</th><th>Our process model</th><th>Y.D. Lin et al. [29]</th><th>Remco et al. [24]</th></tr>
<tr><td rowspan="3">Visibility</td><td>Task Data</td><td rowspan="2">Supported</td><td rowspan="3">Unsupported</td><td rowspan="7">N/A</td></tr>
<tr><td>(Sub)Process Data</td></tr>
<tr><td>Multiple Artifact Instance</td><td>Input attribute</td></tr>
<tr><td rowspan="4">Artifact Interaction</td><td>Task to Task</td><td rowspan="4">Distinct control and data channels</td><td rowspan="4">Integrated control and data channels (Global data)</td></tr>
<tr><td>Task to Sub-process</td></tr>
<tr><td>Sub-process to Task</td></tr>
<tr><td>Sub-process to Sub-process</td></tr>
</table>

7.2. Advantages of H_c^T PNets

When a process is modeled with a PNet, CPNet or Timed CPNet, the behavior of the WfMS, on which the process executes, may not be included. Thus, the behavior simulated upon the nets may not indicate the behavior of real WfMS. And, the analysis results gained upon the nets might be useless. The problems can be solved partially with H_c^T PNets. For example, many correlations between the artifact/process and its operations cannot be found in above nets, but in H_c^T PNets.

In addition, H_c^T PNets can represent a BPMN-based process with a sub-process which is associated with a lower-level net, especially for *Standard* and *MultiInstance* loop sub-process. The refinement function is not supported by PNet, CPNet and Timed CPNet.

Table 7.4 Advantages of H_C^T PNets

H_C^T PNets	PNets\ CPNets\ Timed CPNets	
Hierarchical Token (Net within Net)	Interactions between WfMS and participants are not captured	All
	High difficulty of maintaining correlations between an artifact state transition and its operations	All
Hierarchical Transition	Un-introduce element refinement mechanism	All
Time Semantic	Time Condition Omission	PNets\ CPNets
Data Semantic	Weak Data Presentation	PNets

Chapter 8. Conclusion and Future Works

Current analysis techniques based on PNets, CPNets, and timed CPNets are not well for workflow modeled with BPMN. The main contribution of this thesis is to introduce a BPMN-based process model which provides an easier way to extract knowledge from the role, control flow, data flow and message flow of a workflow. Such a BPMN-based can be transformed into a H_C^T PNets, which is an extended timed CPnets with hierarchical token, for analysis.

The BPMN process may include: 1) an interaction between participants, 2) a multi-instance (loop) activity, 3) an event-triggered (supplement) process, 4) a join node designed by one of the three advanced join mechanisms, *discriminator, multiple merge* and *N out of M join*, and 5) a data flow described with explicit channel. The analysis for H_C^T PNets works for BPMN workflow of well-formed or unstructured control flows.

We currently continue our research in several directions. First, we plan to implement our model and methods on existing workflow management systems, such as Microsoft Visio [25] or BizAgi BPM [26], in order to apply our research result in real-world applications. The second is to continue the research of analysis on activities (task and sub-process) or process instances with more complex events. Thirdly, we plan to integrate our resource constrains analysis techniques to develop a design methodology for constructing workflows or web services.

Reference

[1] The Workflow Management Coalition, "*The workflow reference model*", Document Number TC00-1003, January 1995.

[2] Object Management Group. *Business Process Modeling Notation* (*BPMN*) Version 1.2. OMG Final Adopted Specification. Object Management Group, 2009.

[3] R. Dijkman, M. Dumas, and C. Ouyang. "Semantics and Analysis of Business Process Models in BPMN.", *Information and Software Technology*, Vol.50, Issue 12, pp. 1281-1294, 2008.

[4] C. Ouyang, M. Dumas, A.H.M. ter Hofstede, W.M.P. van der Aalst, "Pattern-based translation of BPMN process models to BPEL Web services.", *Journal of Web Services Research*, Vol.5, No. 1, pp. 42-62, 2007.

[5] C. Ouyang, M. Dumas, A.H.M. ter Hofstede, and W.M.P. van der Aalst, "From BPMN process models to BPEL Web services.", *Proceedings of the 4th International Conference on Web Services*, pp. 285-292, Chicago, Illinois, USA, Sep., 2006.

[6] S. White. Using BPMN to Model a BPEL Process. BPTrends, 3(3):1–18, March 2005.

[7] S. Sadiq, M.E. Orlowska, W. Sadiq, and C. Foulger, "Data flow and validation in workflow modeling.", *Proceedings of the 15th Australasian database conference*, pp. 207-214, Dunedin, New Zealand, January 2004.

[8] S.X. Sun, and J.L. Zhao, "A data flow approach to workflow design.", *Proceedings of the 14th Workshop on Information Technology and Systems*, pp. 80-85, 2004.

[9] S.X. Sun, J.L. Zhao, and O.R. Sheng, "Data flow modeling and verification in business process management.", *Proceedings of the AIS Americas Conference on Information Systems*, pp. 4064-4073, New York, August 5-8, 2004.

[10] S.X. Sun, J.L. Zhao, J.F. Nunamaker, and O.R.L. Sheng, "Formulating the data

flow perspective for business process management.", *Information Systems Research*, Vol. 17, No. 4, pp. 374-391, December 2006.

[11] C.H. Wang and F.J. Wang, "Detecting artifact anomalies in business process specifications with a formal model.", *Journal System and Software*, 2009.

[12] S. Ha, H.W. Suh, "A timed colored Petri nets modeling for dynamic workflow in product development process.", *Computers in Industry*, Vol.59, Issue 2-3, 2008.

[13] W. Shen et al., "Hierarchical Timed Colored Petri Nets Based Product Development Process Modeling.", *CSCWD2004*, *LNCS* 3168, pp. 378 – 387, 2005.

[14] K. Jensen, "Coloured Petri nets: Basic concepts, Analysis methods, and Practical use", volume 1-3, Springer-Verlag, 1992.

[15] P. Bouyer, S.Haddad and P.-A.Reynier, "Timed Petri nets and timed automata: On the discriminating power of zeno sequences", *Information and Computation*, Vol. 206, Issue 1, Jan., 2008.

[16] P. Merlin and D. J. Farber, "Recoverability of communication protocols – implication of a theoretical study.", *IEEE Trans. on Communications*, Vol. 24, Issue 9, pp. 1036-1043, 1976.

[17] B. Walter. "Timed Petri nets for modeling and analyzing protocols with real time characteristics.", *Proceedings of the 3rd IFIP Workshop on Protocol specification, Testing, and Verification*, pp. 149-159. North Holland, 1983.

[18] C. Ramchandani, "Analysis of asynchronous concurrent systems by timed Petri nets.", Technical Report MAC-TR-120, Massachusetts Institute of Technology, February 1974.

[19] J. Coolahan and N. Roussopoulos, "Timing requirements for time-driven systems using augmented Petri nets.", *IEEE Trans. on Software Eng.*, Vol. SE-9, No. 5, pp. 603-616, 1983.

[20] P. A. Abdulla and A. Nylen, "Timed Petri nets and BQOs.", *Proceedings of the 22nd Int. Conf. on Applications and Theory of Petri Nets*, Vol. 2075 of LNCS, pp. 53-70, Springer-Verlag, 2001.

[21] W. van der Aalst, "Interval timed coloured Petri nets and their analysis.",

Proceedings of the 14th Int. Conf. on Applications and Theory of Petri Nets, Vol. 961 of LNCS, pp. 452-472, Springer-Verlag, 1993.

[22] K. Jensen, "Coloured Petri Nets Basic Concepts, Analysis Methods and Practical Use.", Vol. 1, Springer-Verlag, 1997.

[23] K. Jensen, "Coloured Petri Nets Basic Concepts, Analysis Methods and Practical Use.", Vol. 2, Springer-Verlag, 1997.

[24] R. M. Dijkman, M. Dumas and C. Ouyang, "Formal Semantics and Analysis of BPMN Process Models.", *Information and Software Technology*, 2009.

[25] Business Process Modeling mit Process Modeler for Microsoft Visio, http://itp-commerce.com/?gclid=CLKNlIGTspsCFcktpAodBzNJNw.

[26] BizAgi BPM software, http://www.bizagi.com/.

[27] C. A. Petri, "Kommunikation mit Automaten," PhD thesis, University of Bonn, Bonn, Germany, 1962.

[28] C. Girault and R. Valk, "Petri Nets for Systems Engineering.", Vol. 1, Springer-Verlag, 1998.

[29] Ou-Yang and Y.D. Lin, "BPMN Based Business Process Model Feasibility Analysis:a Petri Net Approach," *International Journal of Production Research*, Vol.46, No.14-15, July, 2008.

[30] W.M.P. van der Aalst, "The application of petri nets to workflow management", *Journal of Circuits, Systems and Computers*, Vol. 8, No. 1, pp. 21-66, 1998.

[31] W.M.P. van der Aalst and A.H.M. ter Hofstede, "Verification of workflow task structures: a petri-net-based approach", *Information Systems*, Vol. 25, No. 1, pp. 43-69, 2000.

[32] H.M.W. Verbeek and W.M.P. van der Aalst, "Woflan 2.0: a petri-net-based workflow diagnosis tool", *Proceedings of the 21st International Conference of Application and Theory of Petri Nets* (ICATPN 2000), pp. 475-484, Aarhus, Denmark, June 26-30, 2000.

[33] N.R. Adam, V. Atluri, and W.K. Huang, "Modeling and analysis of workflows

using petri nets", *Journal of Intelligent Information Systems*, Vol. 10, No. 2, pp. 131-158, March/April 1998.

[34]R. M. Dijkman, M. Dumas, and C. Ouyang. Formal Semantics and Analysis of BPMN Process Models using Petri Nets. Technical report, Faculty of Information Technology, Queensland University of Technology, 2007.

[35] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow patterns", BETA Working Paper Series, WP 47, Eindhoven University of Technology, Eindhoven, 2000.

[36] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Advanced workflow patterns", *Proceeding of 7th International Conference on Cooperative Information Systems* (CoopIS 2000), Vol. 1901 of Lecture Notes in Computer Science, pp. 18-29. Springer-Verlag, Berlin, 2000.

[37] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst, "Workflow data patterns", QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[38] J. Bae, H. Bae, S.-H. Kang, and Y. Kim, "Automatic control of workflow processes using ECA rules", *IEEE Transaction on Knowledge and Date Engineering*, Vol. 14, No. 8, pp. 1010-1023, IEEE Computer Society, August 2004

[39] Aalst van der, W. M. P., "Verification of WF-nets," Application and Theory of Petri Nets, Vol. 1248 of Lecture Notes in Computer Science, 1997.

[40]W.M.P. van der Aalst. "Three Good Reasons for Using a Petri-net-based Workflow Management System." *Proceedings of the International Working Conference on Information and Process Integration in Enterprises* (IPIC'96), pages 179-201, Camebridge, Massachusetts, Nov 1996.

[41]R. Eshuis and J. Dehnert., "Reactive Petri nets for Workflow Modeling," Application and Theory of Petri Nets 2003, Vol. 2679 of Lecture Notes in Computer Science, pp. 295-314. Springer-Verlag, Berlin, 2003.

[42]S. Ha and H. Suh, A timed colored Petri nets modeling for dynamic workflow in product development process, Computers in Industry 59 (2008), pp. 193-209.

[43]W. Penczek and A. Pólrola , "Advances in Verification of Time Petri Nets and Timed Automata .", Springer-Verlag, 2006.

[44]H. J. Hsu and F. J. Wang, "Using Artifact Flow Diagrams to Model Artifact Usage Anomalies.", *3rd IEEE International Workshop on Quality Oriented Reuse of Software*, 2009.

[45] R. Liu, and A. Kumar, "An Analysis and Taxonomy of Unstructured Workflows," BPM2005, pp. 268-284, Nancy, France, September 2005.

[46]PetiaWohed, Wil M.P. van der Aalst, Marlon Dumas, Arthur H.M. ter Hofstede, and Nick Russell. Pattern-based Analysis of BPMN. Technical report, 2005. http://www.bpm.fit.qut.edu.au/projects/babel/docs/BPM-05-26.pdf.

[47] B. Kiepuszewski , A. H. M. ter Hofstede , C. Bussler, "On Structured Workflow Modelling.", *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pp.431-445, June 05-09, 2000.

[48]Ivo Raedts, Marija Petkovic, Yaroslav S. Usenko, Jan Martijn E. M. van der Werf, Jan Friso Groote, Lou J. Somers: "Transformation of BPMN Models for Behaviour Analysis.", MSVVEIS 2007: 126-137.

[49]T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, "Introduction to Algorithms," Second Edition, The MIT Press, 2001.