國立中山大學資訊管理研究所

碩士論文

利用派翠網來塑模和檢驗個人工作流程

The use of Petri Nets to Personal process modeling and verification

研 究 生：陳 鈴 雅　撰

指導教授：黃 三 益 博士

中華民國九十四年七月

致謝詞

從中山資管系到中山資管所，六年的時間在中山欣賞美景，研究所的生涯讓我學到很多，也遇到許多挫折，可是還是完成了這個階段的任務，在心中更是感謝那些給予我鼓勵的人，首先，最想感謝的就是指導教授黃三益博士的教導，在寫論文的這段時間裡，老師做學問嚴謹的態度，平時對我們的照顧，都讓我們銘記在心。

第二個要感謝的是高彬學長，學長將過去的研究累積承傳下來，讓我省去許多不必要的摸索時間。接著要感謝同學們志吉、鎔碩、祐平互相扶持與鼓勵，總是不停的提供歡笑給實驗室，也讓我們看到他們在紓解壓力上的獨特方式。

鈴雅

論文提要

學年度： 93

學期： 2

校院： 國立中山大學

系所： 資訊管理研究所

論文名稱(中)： 利用派翠網來塑模和檢驗個人工作流程

論文名稱(英)： The use of Petri Nets to Personal process modeling and verification

學位類別： 碩士

語文別： 英文

學號： 924020036

提要開放使用： 是

頁數： 63

研究生（中）姓： 陳

研究生（中）名： 鈴雅

研究生（英）姓： Chen

研究生（英）名： Lin-Ya

指導教授（中）姓名： 黃三益

指導教授（中）姓名： San-Yih Hwang

關鍵字(中)： 個人工作流程, 派翠網

關鍵字(英)： Personal workflow, Petri Nets

中文摘要

個人化流程是在協調個人的活動，流程的目的是在使用者和相關組織的限制下達成個人目標。在本論文中，我們改以派翠網來塑模個人工作流程以解決缺少控制流所產生的問題。我們重新定義了個人工作流的正確性，並且提出在派翠網上的檢驗方法。在架構上，我們增加了線上執行引擎讓使用者可以透過網際網路即時的執行和檢驗個人流程的正確性。我們也可以透過個人工作流程系統來管理個人流程，而個人工作流程管理系統是在手持裝置上執行的。因為手持式裝置計算能力和電力的限制，所以只有當使用提出要求時我們才提供個人流程正確性的檢驗。

Abstract

A personal process is a coordination of personal activities, each requiring a joint effort between a user and an enacting organization. In this thesis, we model a personal process using Petri Nets to describe both the control flow and data flow pertaining to the personal process. We redefine the correctness of a personal process and address the verification method based on Petri Nets. In our architecture, we add an online execution engine for the user to execute and verify the correctness of a personal process in real time through the Internet. A personal process can also be managed by a *personal workflow management system* (PWFMS) running on a handheld device. Because of the strict limitations on their computation power and battery consumptions, we support verification only when the wireless connection is available.

Table of Contents

List of figures

List of tables

Chapter 1 Introduction

1.1 Background

In recent years, wireless devices are becoming more powerful and popular. People use many hand-held devices such as the personal digital assistants and cellular phones to communicate with each other. Typical applications include the management of personal resources such as calendars, memos, address books, to-do-list, games, and the access to the many resources available on the Internet via some protocol such as WAP. Nowadays mobile devices are widely used by individuals to communicate and handle personal businesses and finding their way in business applications as well.

1.2 Motivation

People's daily activities are not independent, and they are likely to be process-oriented. We distinguish two types of activities: business activities and personal activities whose objectives are to achieve business and personal goals respectively. In the context of business activities, workflow management systems have been used in enterprises to automate their crucial business processes, each encompasses a set of related business activities. A workflow management system mainly used to coordinate business processes in enterprises. It separates the specification of business processes from their execution and controlling its executions. A workflow management system must have well-formed structures and, once specified, are often instantiated many times. In contrast, a personal process is a coordination of a set of personal tasks so as to achieve a personal goal and each personal process instance often has a unique structure, in other words after a process is specified, only one instance is to be executed [Hwan03]. A personal process is defined as a coordination of a set of

personal tasks, and each task has to be initiated by the user and the goal is to achieve some personal goal. A personal task could be a simple activity or a business process that is posted by an organization and initiated at the request of users. There are many personal processes in our daily lives, such as applying a credit card, filing an entrance application request to a school, and purchasing an insurance policy, each of which may involve a number of personal tasks. Here a personal task is defined as an interaction of a user and an enacting organization. In case of face to face interaction, a personal task can be conducted only under some time and location constraints. For example, paying a bill via an ATM machine can be done only between 8am to 12am and at certain locations. In [Chen01], the author proposed a personal process model and implemented on a personal workflow management system (PWFS) running on Palm PDA based on such a model. This architecture allowed users to define their personal processes and provided a set of operations for manipulating them. Such operations allow the PWFS to passively or proactively provide useful information, such as the set of personal tasks that can be done at the current time and current location, to the user. Figure 1-1 shows the architecture of a personal workflow system.

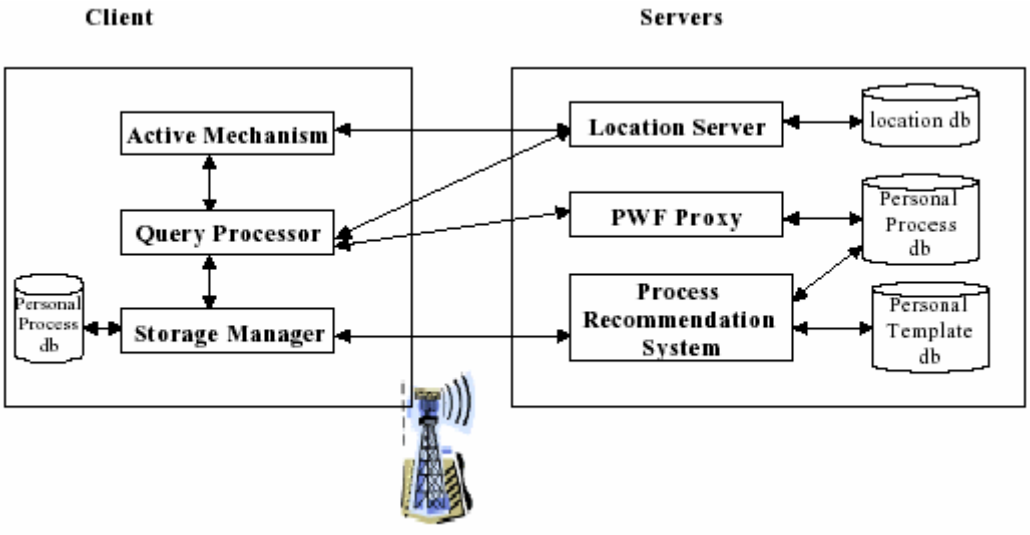

Figure 1-1: Personal Workflow System Architecture

Aside from some unique features exhibited by its personal tasks, such as executable time/places and physical input data items, a personal process sometimes requires the interaction of a user with several organizations. While it is possible to enforce a personal process by an interorganizational WFMS that coordinates tasks executed in several organizations, it is not wise to do so due to the heavy cost associated with the interorganizational WFMS. Thus, we propose a light-weight WFMS specifically designed for designing, analyzing, and enforcing personal processes.

The requirement of the personal workflow system can be characterized as follows:

- For a given user, a personal process can be executed for only a limited number of times (and only once in most cases). Personal tasks are primarily related and finished by their executable times, executable places, and input and output data.
- A personal task, in most cases, is an interaction between a user and an enacting organization. While the operation of the organization is beyond the control of the user, it is important that the user is notified about the status of the task.
- One important goal of managing personal processes is to remind or to provide suggestions to a mobile user. Query capabilities of a Personal Workflow Management System (PWFMS) that keep a mobile user updated about the current personal process status are essential.
- The Personal Workflow Management System controls the data flow. In each task it may need the specified input data items and output some data items which serve the input of the next task. This is a data dependency relation.
- There are constraints that can be used to verify the aliveness of the process. If the process is not alive, there is no need to continue executing it.

However, there are shortcomings in the proposed personal process model:

1. There is no control flow to handle the process. While it is acknowledged that users seldom impose rigid rules on the execution sequence of personal tasks, specification on control flow is still needed in some cases.
2. It lacks a unified theoretical model and analysis approach. The current personal workflow model is ad-hoc and hard to generalize. A theoretical model also allows for the verification of a personal process in a graceful way.

Consider the following personal process: the student registration process currently adopted at NSYSU. A student first registers courses on the Internet. Some time later she will receive an invoice for payment and may decide to pay on the ATM machine or at the counter of the bank during its open time. However, if she forgets to pay it in time she can only pay the course fees by going to the bank. After paying the course fee, she will receive a tuition receipt. Then she can bring the receipt with her to the office of the academic affair and receive a stamp on her student id card. Note that most of the execution orders are determined by the data dependency between tasks. The only exception is that between “Pay course fee in time” and “Pay late course fee”. They are exclusive in the sense that the completion of one activity will invalidate the other. Unfortunately, the exclusive selection construct in not included in the current personal process model.

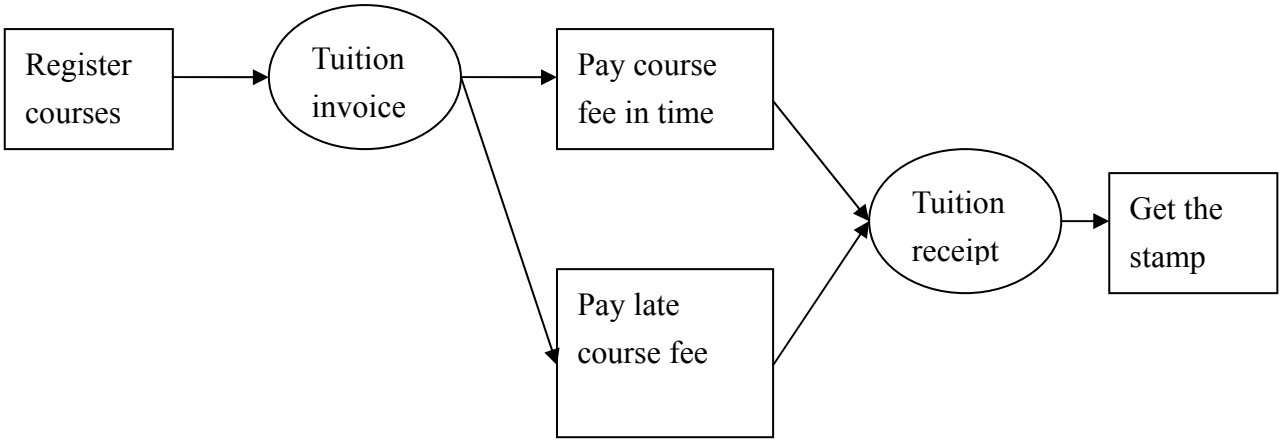

Figure 1-2: Student registration process

The existing personal process model adopts the metagraph to describe the data dependency in the process. Metagraph is a graphical tool mainly designed for modeling data flow and it lacks some expressive power such as control flow modeling. Furthermore, the analytic tool to the model is limited. In [Kao04], a number of constraints have been proposed for personal processes, and some of them cannot be directly verified by the existing analysis tools.

Our contribution in this thesis includes the following:

1. We adopt a theoretical model, namely timed Petri Nets, as the underlying model for personal processes. This model allows us to describe both control and data dependencies in a coherent way. The algorithm for mapping from high level personal process to timed Petri Nets is described.
2. We redefine the correctness of personal processes and propose an algorithm for correctness verification.
3. We implement a prototyped PWFS based on the proposed model and algorithm for concept proof.

1.3 Thesis organization

The rest of this thesis is organized as follows. In Chapter 2, we review some related work. In Chapter 3 we formally define the personal process model. In Chapter 4 we describe an approach for mapping the work flow to Petri Nets and define the correctness criteria for personal processes. Chapter 5 presents algorithms for correctness verification, and Chapter 6 describes our implementation effort. Finally, Chapter 7 summarizes this thesis and identifies future research directions.

Chapter 2 Literature review

We first review previous work in modeling and managing personal processes. We then introduce some fundamental concepts of Petri Nets that we will use to describe the control and data flow in the proposed personal process model.

2.1 Personal workflow

In [Chen01], Chen proposed a personal process model and an algebra that includes a set of operations for inquiring a personal process. The definition of each operation is listed in Table 2-1(T means a set of tasks and D a set of data):

Table 2-1: Operations defined in [Chen01]

Notation	Operation name	Type
$\circ_p$	***PLACE_OVERLAP***	$T \times T \rightarrow T$
	$S_1 \circ_p S_2 = \{t1 : t1 \in S_1, (\exists t2 \in S_2, \exists p1 \in t1.p, \exists p2 \in t2.p, RECTANGLE\_OVERLAP(p1, p2))\}$	
	Produces a subset of S_1 whose executing place overlaps with the executing place of some task in S_2.	
$\circ_t$	***TIME_OVERLAP***	$T \times T \rightarrow T$
	$S_1 \circ_t S_2 = \{t1 : t1 \in S_1, (\exists t2 \in S_2, \exists i1 \in t1.t, \exists i2 \in t2.t, INTERVAL\_OVERLAP(i1, i2))\}$	
	Produces a subset of S_1 whose executing time overlaps with the executing time intervals of some task in S_2.	
$\xrightarrow[dt]{}$	***MAKE_EXECUTABLE***	$D \times T \rightarrow T$
	$D \underset{dt}{\rightarrow} T = \{t : t \in T, t.i \subseteq D\}$	
	Returns a subset of T, each of which has the input data as a subset of D.	
$\xrightarrow[dd]{}$	***NEED_TASK***	$D \times D \rightarrow D$

	$MinMetaPath(D_1, D_2) = 0$ if $D_2 \subseteq D_1$ $MinMetaPath(D_1, D_2) = \underset{Output(T')=D_2}{Minimum}(MinMetaPath(D_1, Input(T')) + Cost(T'),$	
	Produces a set T of tasks that can be collectively executed by taking D_1 as the input and producing D_2 and has the lowest cost.	
$\uparrow_i$	***COMBINED_INPUT***	$T \rightarrow D$
	$\uparrow_i (T) \equiv \{\cup t.i : t \in T\} - \{\cup t.o : t \in T\}$	
	Returns a set of data, each of which is an element of input data of some element in T but not in the output data of any element in T.	
$\uparrow_o$	***COMBINED_OUTPUT***	$T \rightarrow D$
	$\uparrow_o (T) \equiv \{\cup t.o : t \in T\}$	
	Returns a set of data, each of which is an element of output data of some task in T.	

Along with the above operations, they also defined set operations, such as *intersection* ($\cap$), *union* ($\cup$), and *difference* ($-$), and a relational operator *selection* (σ) for users used in querying task. In additional, a declarative query language similar to SQL-statements was also proposed.

In [Lin02], Lin extended the above-mentioned system architecture by adding a personal process template provider, which allows the designer to define various templates for a personal process and enable users to choose a template that meets their interests and background. This module aims to relieve the user from the tedious work of defining a personal process. In [Tu03], Tu extended the model by introducing task's threads and sets of target data, and define correctness for personal processes. In addition, their work adopted Web services for communication between different components in a PWFS. In [Kao04], the correctness of a personal process is further extended to consider some QoS measures, like execution time or cost becomes too

high, the reliability becomes too low, etc. When a personal process becomes incorrect, continuing the process execution is a waste of resources, and a notification should be made as early as possible.

2.2 Petri Nets

Petri Nets is a graphical modeling tool as well as a mathematical tool. It has a wide range of applications. As a graphical tool, Petri Nets consists of a set of graphical constructs similar to flow charts, block diagram and network and allows for visual communication. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems. The classical Petri Nets was invented by Carl Adam Petri in the sixties ([Petr62]), and its research and extension have considerably increased since then. The classical Petri Nets is a directed bipartite graph with two types of nodes, namely *places* and *transitions*. The nodes of different types are connected via directed *arcs*. Places are often represented by circles and transitions by rectangles.

Table 2-2: Formal Definition of a Petri Nets [Mura89]

$P=\{p_1,p_2,\ldots,p_m\}$	is a finite set of places,
$T=\{t_1,t_2,\ldots,t_n\}$	is a finite set of transitions,
$F \subseteq (P \times T) \cup (T \times P)$	is a set of arcs (flow relation),
$W: F \rightarrow \{1, 2, 3, \ldots\}$	is a weight function,
$M_0: P \rightarrow \{0, 1, 2, 3, \ldots\}$	is the initial marking,
$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$	

Formally, a Petri Net *PN* is a 4-tuple (*P*, *T*, *F*, *W*), whose definitions are shown in Table 2-2. A place *p* is called an *input place* of a transition *t* iff there exists a directed arc from *p* to *t*. Place *p* is called an *output place* of transition *t* iff there exists a

directed arc from t to p. We use $\bullet t$ to denote the set of input places for a transition t. The notations $t\bullet$, $\bullet p$ and $p\bullet$ have similar meanings, e.g., $p\bullet$ is the set of transitions sharing p as an input place.

At any time a place contains zero of more tokens, drawn as black dots. The state, often referred to as marking, is the distribution of tokens over places. The number of tokens may change during the execution of the net. Transitions are the active components in a Petri Net, and they change the state of the net according to the following firing rule:

(1) A transition t is said to be *enabled* if each input place p of t is marked with at least $w(p,t)$ tokens. Where $w(p,t)$ is the weight of the arc from p to t.

(2) An enabled transition may or may not fire(depending on whether or not the event actually takes place).

(3) A firing of a enabled transition t removes $w(p,t)$ tokens from each input place p of t, and adds $w(p,t)$ tokens to each output place p of t , where $w(p,t)$ is the weight of the arc from t to p.

Figure 2-1shows a simple Petri Net, consisting of three places (*claim*, *under_consideration* and *ready*) and three transitions (*record*, *pay* and *send_letter*). This network models the process for dealing with an insurance claim. Arriving at the place *claim*, it is first recorded, after which either a payment is made or a letter sent explaining the reasons for rejection.

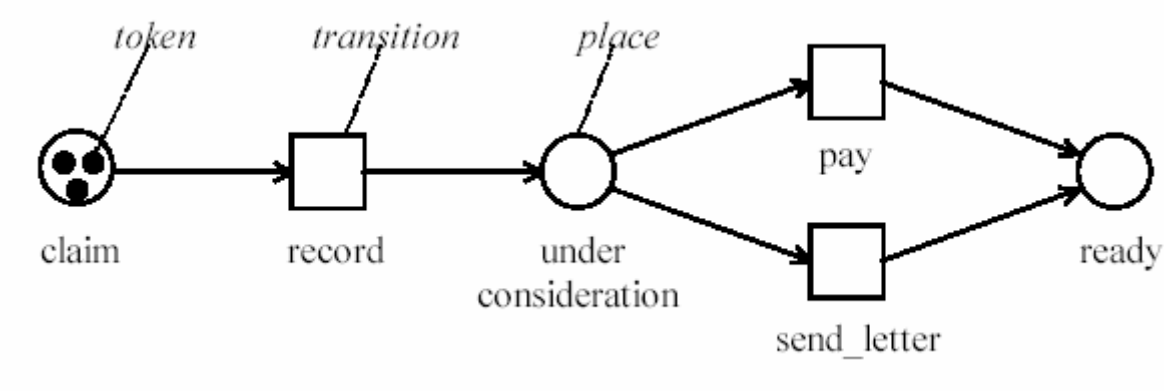

Figure 2-1: A classic Petri Net

Modeling a workflow process definition in terms of a Petri Net is rather straightforward: tasks are modeled by transitions, conditions are modeled by places, and cases are modeled by tokens. A Petri Net which models a workflow process definition (i.e. the life-cycle of one case in isolation) is called a *WorkFlow net* (WF-net) [Aals98]. A WF-net satisfies two requirements. First of all, a WF-net has one input place (*i*) and one output place (*o*). A token in *i* corresponds to a case which needs to be handled, a token in *o* corresponds to a case which has been handled. Secondly, in a WF-net there are no dangling tasks and/or conditions. Every task (transition) and condition (place) should contribute to the processing of cases. Therefore, every transition *t* (place *p*) should be located on a path from place *i* to place *o*. The latter requirement corresponds to strongly connectedness if *o* is connected to *i* via an additional transition *t*.

2.2.1 Routing sequence

A workflow process definition specifies how the cases are routed along the tasks that need to be executed. Figure 2-2, 2-3, 2-4, 2-5 shows the routing constructs identified by the Workflow Management Coalition (WfMC). The WfMC was founded in 1993 and in January 1995 the WfMC released a glossary which provides a common set of

terms for workflow vendors, end-users, developers, and researchers ([WFMC96]). In this glossary, four types of routing are identified:

- *sequential*

  Tasks are executed sequentially if the execution of one task is followed by the next task. In Figure 2-2 task *B* is executed after task *A* has been completed and before task *C* is started.

- *parallel*

  In Figure 2-3 task *B* and task *C* are executed in parallel. This means that *B* and *C* are executed at the same time or in any order. To model parallel routing, two building blocks are identified: (1) the *AND-split* and (2) the *AND-join*. The AND-split in Figure2-3 enables *B* and *C* to be executed after *A* has been completed. The AND-join synchronizes the two parallel flows, i.e., task *D* may start after *B and C* have been completed.

- *conditional*

  In Figure 2-4 either task *B* or task *C* (exclusive OR) is executed. To model a choice between two or more alternatives we use two building blocks: (1) the *OR-split* and (2) the *OR-join*. If task *A* is executed, a choice is made between *B* and *C*. Task *D* may start after *B or C* is completed.

- *iteration*

  Sometimes it is necessary to execute a task multiple times. In Figure 2-5 task *B* is executed one or more times.

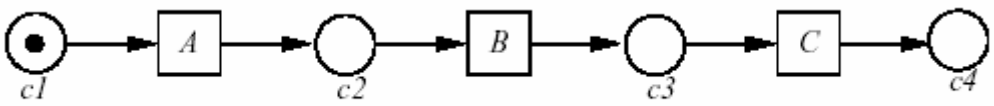

Figure 2-2: Sequential routing

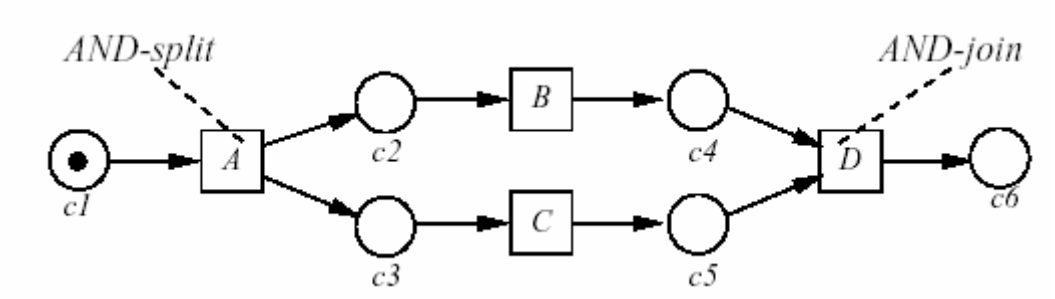

Figure 2-3: Parallel routing

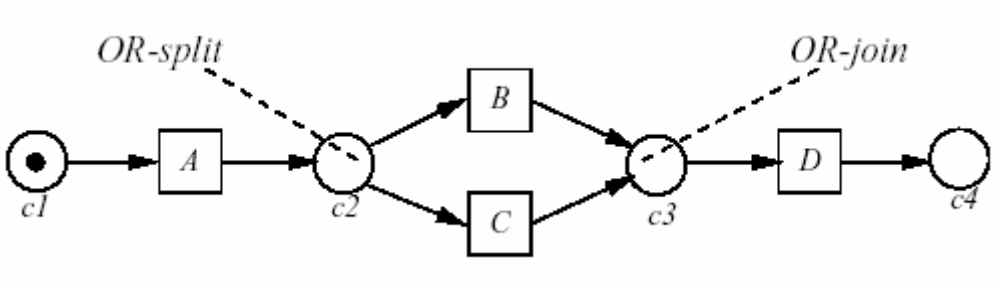

Figure 2-4: Condition routing

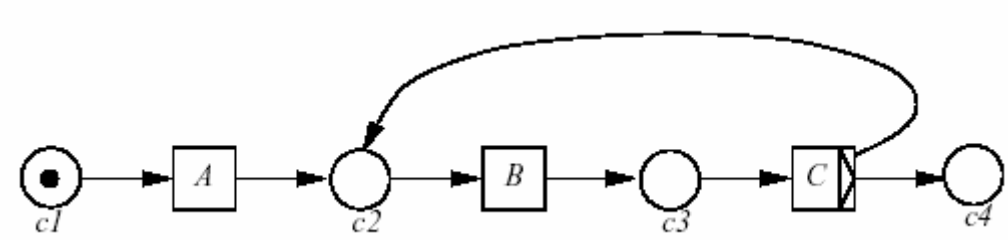

Figure 2-5: Iteration routing

These four types of constructs are basic components for the construction of so-called *structured* workflows [Kiep00].

2.2.2 Properties

A major strength of Petri Nets is their support for analysis of many properties and problems associated with concurrent systems. In this section, we will first describe some properties related to the workflow system. For simple Petri Nets, the length of time between enabling a transition to its firing is indeterminate. Firing a transition is assumed to be an instantaneous event. Firing is also non-deterministic which means

that enabled transitions are never forced to fire.

1. Liveness

Liveness usually means the complete absence of deadlocks. A Petri Net is said to be live if, no matter what marking has been reached, it is possible to fire any transition in the net by progressing through some further firing sequence. This means that a live Petri Net guarantees deadlock-free operation of the modeled system.

There are other concepts related to liveness such as liveness levels. A transition *t* can be categorized as follows:

L0-live (dead): if *t* can never be fired in any firing sequence.

L1-live (potentially friable): if *t* can be fired at least once in some firing sequence.

L2-live: For given any integer *n*, there exists a firing sequence where the firing of *t* occurs at least *n* times.

L3-live: There exists an infinite firing sequence in which *t* occurs infinitely often.

2. Reachability

Reachability is a fundamental basis for analyzing the dynamic properties of any Petri Net system. Reachability points out whether or not a system can reach a specific state or behavior. The firing of an enabled transition will change the markings; a sequence of firings will result in a sequence of markings. That is given a Petri Net *PN*, a marking M_k is reachable if there exists a sequence of firings that will lead M_0 to M_k. This expression can be denoted by $M_k \in R(M_0)$.

3 Safeness

A place in a Petri Net is safe if the number of tokens in that place never exceeds one. A Petri Net is safe if all places in the net are safe.

4 Boundedness

By examining whether or not a net is bounded and safe, it can be predicted that there

will be overflows in buffers or registers. A Petri Net (PN, M_0) is said to be k-bounded or simply bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0.

5 Coverability

A marking M in a Petri Net (N,M_0) is said to be coverable if there exists a marking M' in $R(M_0)$ and each number of token in each place in M' is bigger than that in M. Coverability is closely related to L1-liveness (potential firability). Let M be the minimum marking needed to enable a transition t. Then t is dead (not L1-live) if and only if M is not coverable. That is , t is L1-live if and only if M is coverable.

6 Reversibility and Home state

A Petri Net is said to be reversible if, for each marking M in $R(M_0)$, M_0 is reachable from M. In other words, a reversible Petri Net can always get back to its initial marking. In many modeling applications, it may be desirable to go back to some (home) state instead of going back to the initial state. In this case, the reversibility condition can be relaxed to a home state.

7 Persistence

In contrast with conflict, where for any two enabled transitions the firing of any one will disable the other, persistence can be thought of as a conflict-free property. A Petri Net is said to be persistent if, for any two enabled transitions, the firing of any one will not disable the other. In other words, the persistence property guarantees that once a transition is enabled, it will stay enabled until it fires. Persistence property is useful in modeling parallel programming and speed-independent asynchronous circuits.

8 Synchronic Distance

Synchronic distance is a measurement used to determine the degree of mutual dependence between two events in a condition/event system. The synchronic distance

between any two transitions can be obtained as follows: Given any two transitions *t1* and *t2* in an initial marked Petri Net PN = {P, T, I, O, M_0}, the synchronic distance is defined as

$$d12 = \max_d |\sigma(d, t1) - \sigma(d, t2)|$$

where *d* is any firing sequence starting from any marking *M* in *R*(*M0*) and σ(*d*, *ti*) is the number of times that transition *ti*, i = 1 or 2 fires in *d*.

- **Extensions of the Petri Nets**

The classical Petri Nets allows for the modeling of states, events, conditions, synchronization, parallelism, choice, and iteration. However, Petri Nets used to describe the real processes tend to be complex and extremely large. Moreover, the classical Petri Nets does not allow for the modeling of data and time. To solve these problems, many extensions have been proposed. Three well-known extensions of the basic Petri Nets model are: (1) the extension with color to model data, (2) the extension with time, and (3) the extension with hierarchy to structure large models.

- extension with color

  Tokens often represent objects (e.g. resources, goods, humans) in the modeled system. However, in the classic Petri Net it is impossible to distinguish between two tokens: two in the same place are by definition indistinguishable. If an insurance claim is modeled by a token in the Petri Nets, we want to represent attributes such as the name of the claimant, identification number, date, and amount. In order to describe an object's characteristics with the corresponding token, the classic Petri Nets is extended using 'color'. This extension ensures that each token is provided with a value or color. A token representing a particular car will, for instance, have a value which makes it possible to identify its make, registration number, year of manufacture, color and owner. We can notate a possible value for such a token as follows:

[brand: 'BMW'; registration: 'J 144 NFX'; year: '1995'; color: 'red'; owner: 'Johnson']

Transitions determine the values of the produced tokens on the basis of the values of the consumed tokens, i.e., a transition describes the relation between the values of the ‘input tokens’ and the values of the ‘output tokens’. It is also possible to specify ‘preconditions’ which take the colors of tokens to be consumed into account.

- extension with time

For real systems it is often important to describe the temporal behavior of the system, i.e., we need to model durations and delays. Since the classical Petri Nets is not capable of handling quantitative time, a timing concept is added. There are many ways to introduce time into the Petri Nets. Time can be associated with tokens, places, and/or transitions. There are two main types of time extension to PNs. One is timed Petri Nets and another is Time Petri Nets [Nidd94]. Difference between these two types is the presentation of the time interval. The time interval in timed Petri Nets means the earliest time to finish the job. The time interval in Time Petri Nets means the last time to finish the job.

- extension with hierarchy

Although timed colored Petri Nets allow for a succinct description of many business processes, precise specifications for real systems have a tendency to become large and complex. In order to structure a Petri Net hierarchically, a new 'building block': a double-bordered square, was introduced in [Aals94][Jens96]. We call this element a *process*. It represents a subnetwork comprising places, transitions, arcs and subprocesses. Because a process can be constructed from

subprocesses, which in turn can also be constructed from (further) subprocesses, it is possible to structure a complex process hierarchically. In order to illustrate this, we shall use a process for dealing with technical faults in a product department in Figure 2-6. Every time a fault occurs - for example, a jammed machine - it is categorized by the department's mechanic. The fault can often be put right as it is being categorized. If this is not the case, then a repair takes place. After this has been done, a test is carried out, with three possible results: (1) the fault has been solved; (2) a further repair is required; or (3) the faulty component must be replaced. Extends with hierarchy can be used to structure large processes. At one level we want to give a simple description of the process (without having to consider all the details). At another level we want to specify a more detailed behavior. The extension with hierarchy allows for such an approach.

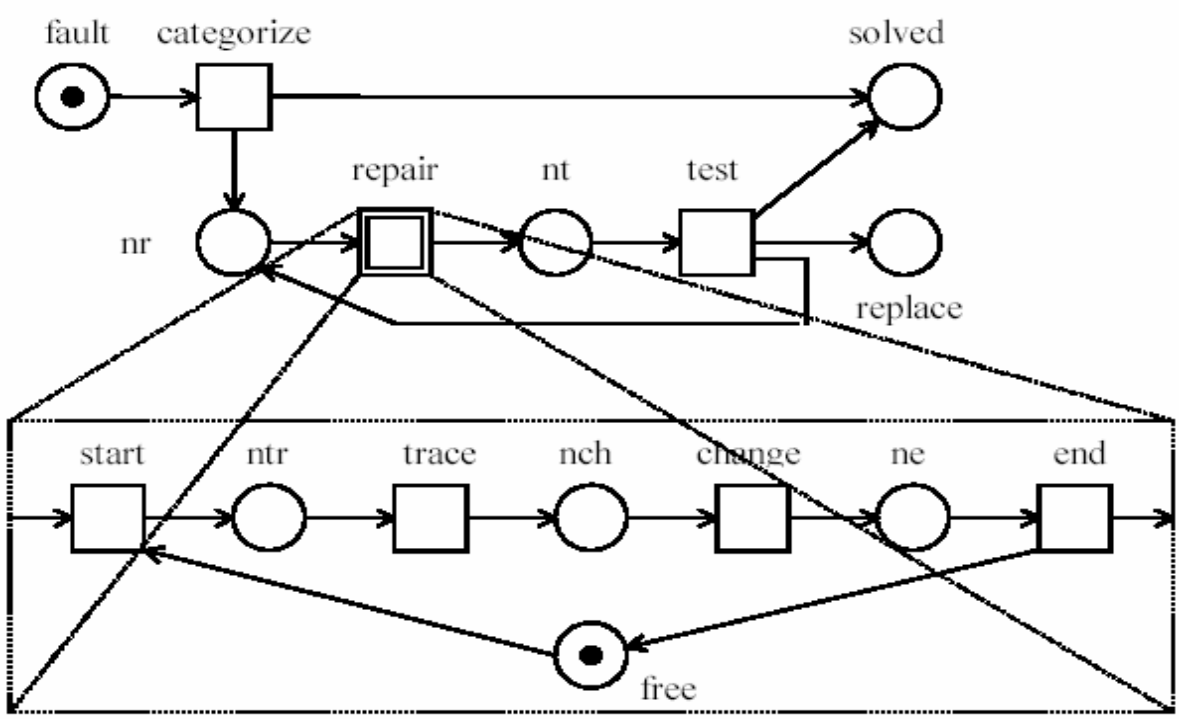

Figure 2-6: The process “solve fault” contains one sub process “repair”

2.2.3 Subclasses of Petri Nets

This part will introduce some Petri Net subclasses. The major advantage of this classification is to increase the modeling power for Petri Nets. The characteristics in these subclasses can be used to determine if a given Petri Nets is a member of the specified subclass.

1. Ordinary Petri Nets (OPN)

   A Petri Net is called ordinary when all of its arc weights are one.

2. State Machine (SM)

   A SM is an ordinary Petri Nets such that each transition has exactly one input place and one output place, i.e., $|\bullet t| = |t\bullet| = 1$ for all $t \in T$.

3. Marked Graph (MG)

   A MG is an ordinary Petri Nets such that each place has exactly one input transition and one output transition, i.e., $|\bullet p| = |p\bullet| = 1$ for all $p \in P$.

   In an SM, transitions have only one input and one output place. In an MG, places have only one input and one output transition. They are also dual from a modeling point of view. A SM can be used to model a conflict situation such that one place has multiple outgoing arcs but cannot represent a concurrent activity or the waiting of synchronization such that one transition has multiple outgoing arcs. An MG, in contrast, can represent concurrency and synchronization but cannot model conflict or data dependent decisions.

4. Free Choice Net (FC)

   A FC is an ordinary Petri Net such that each arc from a place is either a unique output of a place or a unique input to a transition, i.e., $|p\bullet| \leqq 1$ or $\bullet(p\bullet) = \{p\}$ for all $p \in P$.

5. Extended Free-Choice Net (EFC)

An EFC is an ordinary Petri Net such that $p1\bullet \cap p2\bullet \neq \varnothing \Rightarrow p1\bullet = p2\bullet$ for all $p1$, $p2 \in P$.

6. Asymmetric Choice Net (AC)

An AC is an ordinary Petri Net such that $p1\bullet \cap p2\bullet \neq \varnothing \Rightarrow p1\bullet \subseteq p2\bullet$ or $p1\bullet \supseteq p2\bullet$ for all $p1, p2 \in P$.

2.2.4 Analysis methods

- **the reachability tree**

Reachability analysis techniques, such as the use of reachability trees, explore all state spaces. Since this is an exhaustive method, it can only be used for bounded systems and analysis of large systems would be extremely challenging.

Given an initial marking M_0 for a Petri Net PN, a sequence of firing of enabled transitions will result in a new marking. Further firing of enabled transitions results in newer markings. To represent these marking changes starting from M_0 (or the root), a tree structure called the reachability tree can be drawn. With this reachability tree, markings starting from the initial marking can be represented by nodes. Each arc represents the firing of an enabled transition that will transform the net from one marking to another. To simplify the representation of a net, the symbol ω is used to represent an infinite set of values. For any integer $n > \omega$, $n \pm \omega = \omega$ and $\omega^3 \geq 0$. The physical meaning is: If a component of a covering marking is ω, then there exists a "loop" in the path from the root to a particular marking. If one covering marking contain more than one, it may indicate a "loop" interaction among the marking changes. The reachability tree can be used to address safeness, boundedness problems. However, the use of ω

downplays its analytical power due to a loss of information. It cannot be used to solve the reachability or liveness problems nor can it determine possible firing sequences.

■ Incidence matrix

For a Petri Net PN with m transitions and n places, the incidence matrix $\boldsymbol{A} = [aij]$ is an $n \times m$ matrix of integers and its typical entry is given by:

$aij = aij^{+} - aij^{-}$

where $aij^{-} = w(i, j)$ is the weight of the arc from transition i to its output place j, and $aij^{+} = w(j, i)$ is the weight of the arc to transition i from its input place j. The physical meaning for aij is the number of changed tokens for place j each time transition i fires. Thus, aij^{+} represents the number of tokens removed and aij^{-} represents the number of tokens added when transition i fires. Therefore, each row of $\boldsymbol{A}$ corresponds to a transition while each column corresponds to a place. It is easy to see from the transition rule that, aij^{-}, aij^{+}, and aij, respectively, represent the number of tokens removed, added, and changed in place j when transition i fires once. As figure 4-1 shows the $aij^{+} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $aij^{-} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$, and $aij = \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$

In the Incidence matrix, the initial marking can be represented as one $1 \times n$ matrix, that represents all the places state. And we also can represent the target marking as a matrix, such as $M_0 = [1, 0, 0, 0]$, $M_n = [0, 0, 0, 1]$, then we can use the equation $M_n = M_0 + x \cdot aij$ to solve if there exist a firing sequence to the target marking. The Incidence matrix can analyze the reachability property of the Petri Net.

Chapter 3 Personal Process Model

In this Chapter, we describe our personal process model, which is closed to that proposed in [Tu03].

3.1 Personal process model

As mentioned, personal tasks require interactions between a user and enacting organizations. While some of such interactions may occur ubiquitously in the virtual world (i.e., any place and any time), many interactions require the user to be present within certain time periods and/or at certain places. Thus, executable times and executable places are included as attributes of personal tasks in our model. In addition, to execute a personal task, some data items have to be made available. An execution of a personal task takes a set of data items as input and generates another set of data items. In fact, executions of a personal task may lead to more than one kind of output. For example, there are two possible outputs for applying a credit card: a new credit card upon the approval of the application and a rejection letter. We call a possible output pertaining to personal process a *thread*.

Definition 3.1. A personal task *t* is 5-tuple <*id*, *loc_time*, *input*, *participant*, *threads*>, where *id* contains several identifier attribute values (e.g., task name, task organization, URL, etc), *loc_time* records a set of time and place pairs eligible for executing *t*, *input* is a set of data items required for initiating the execution of *t*, *participatant* contains the information about the enacting organization, and *threads* is a collection of data item sets, each representing a possible outcome as the execution of *t*. A thread of a task *t* indicates a possible outcome as the execution of *t*.

Definition 3.2. A personal process is a triple (D, T, C), where D is a set of data items, T is a set of personal tasks, and C represents a set of control transitions between tasks in T. For each task $t \in T$, t.input$\subseteq D$, and $\forall h \in t$.threads, $h \subseteq D$.

Note that there are many models one can use to specify the control transitions between tasks, including Petri Net [Adam98][Aals94][Aals98], State Chart [Wodt97], ECA rules [Daya91][Gepp98], and directed graph [WFMC96]. Various control patterns that exist in commercial workflow management systems or research prototypes have been identified [Aals03]. In our research, we adopt Petri-Nets for specifying the control dependencies of personal processes. As will be shown in Chapter 4, Petri-net can be used to coherently describe both the control and data flow of a personal process, and it also eases the task of correctness verification.

Note that in our work, a data item must exist in some form. In other words, it must be able to be presented to the enacting organization of a task as an input item. In addition, we categorize data items into two types: *primitive* and *processed*. Processed data must be generated by at least (a thread of) one task. Data items that are not processed data are called primitive data. Primitive data may or may not be produced by any task modeled in the system. Primitive data could be a data file, a blank form, a personal belonging (e.g., ID card or credit card), or anything that can be prepared by the user. Processed data are available only when at least one task that is capable of producing it is completed, e.g., a receipt generated by a payment task.

For a personal process instance, we associate a status variable to each data item, each thread, and each task. A data item is in any of the following two states: UNAVAILABLE and AVAILABLE that describes its availability. A thread can be of

the following states: UNDECIDED, FAILED, and SUCCESSFUL. A thread is said to be UNDECIDED if the pertaining task is yet to be finished, SUCCESSFUL if the pertaining task has produced the output data items as indicated in the thread, and FAILED otherwise. A task has six states: INITIAL, READINESS, EXECUTING, COMPLETED, DEAD, and FAILED. Initially, a task is in INITIAL state. When all the data items as needed in the input are AVAILABLE and the current time and place make the task executable, the task is said to be READY. Once the task is executed by the enacting organization, it is said to be EXECUTING. When the task is completed, the task is COMPLETED. If task is expired or it would no be reached by any sequence, it is said to be DEAD. However, if for some reason the task is voluntarily or involuntarily terminated by the user, the task is said to be FAILED. In this case the task behaves as if it was never executed. If the user decides to re-instantiate the task, it will become INITIAL. Figure 3.1 shows the state transition diagram of a task. In the following, when we say a task *t* is *executable*, we actually mean that *t* has not expired (that is, not all executable times are passed) and its state is either INITIAL or READY.

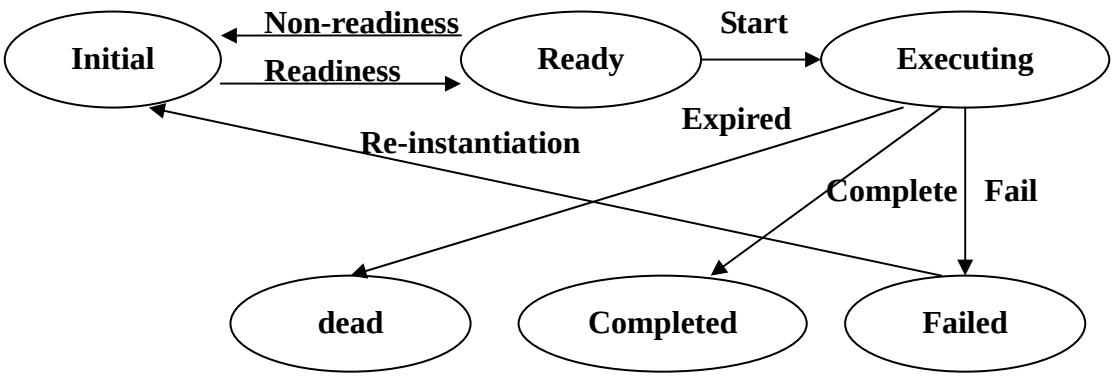

Figure 3-1: State transition diagram of a task

A personal process instance has four states: INITIAL, EXECUTING, COMPLETED, and FAILED. Before any task in a personal process *p* is executed, the personal

process instance is said to be in INITIAL state. Once a task starts execution, *p* is in EXECUTING state. When some desired result is obtained, *p* is said to be COMPLETED. We will define what is meant by desired result in the next section. If, for some reason, the user of *p* decides to abandon *p* before it reaches COMPLETED state, *p* enters FAILED state.

3.2 An example personal process model

In this section, we present the graduating process for an undergraduate student from our university using the proposed model. This example process will be used throughout the entire thesis to illustrate our approaches in executing and verifying personal processes.

To earn a bachelor degree at NSYSU, a student has to meet all the graduation requirements by following a graduation process. Note that depending upon the background and the majors, the graduation processes for different students may differ a bit. In the following, we describe a complete graduation process for a female student. Upon successfully completing the graduation process, a student is awarded an undergraduate degree and subsequently receives her diploma. The first task of the graduation process is to get online and print out a graduation check list. Then she may go to different places for performing different tasks in arbitrary order, which are shown in Table 3-1. The task “Credit check”, which is performed at the office of academic affairs during working hours, checks if the student has earned a minimum number of course credits as required by her major. The result (or output) of this task could be passed (marked as a stamp on the checklist) or failed. “ Departmental check”

is a process for department staff to ensure the student has fulfilled all the requirements imposed by the department. The task “Returning bachelor robe” returns the rented bachelor robe to the office of general affairs. The task” Book return” is conducted at the library to clear the library records of the student. The task “Leaving correspondence information” mandates the student to leave the correspondence information after graduation for future communication. The task “Moving out dormitory” checks if the student has moved out and cleared the bedroom she occupied. The task “Taking English test” is repetitively conducted until the test result satisfies the minimum requirement imposed by the university (TOFLE 213 or TOEIC 650). When enrolled in the university, the student can take English exam at any time. In fact, the task “Departmental check” will also check the score sheet presented by the student. Finally, the task “Receiving diploma” marks the successful completion of the graduation procedure, and the student is given a diploma. The control dependencies of the various tasks are shown in Figure 3.2. Figure 3-3 shows the data dependencies of this process. The example will be illustrated by a Petri Net graph in Chapter 4.

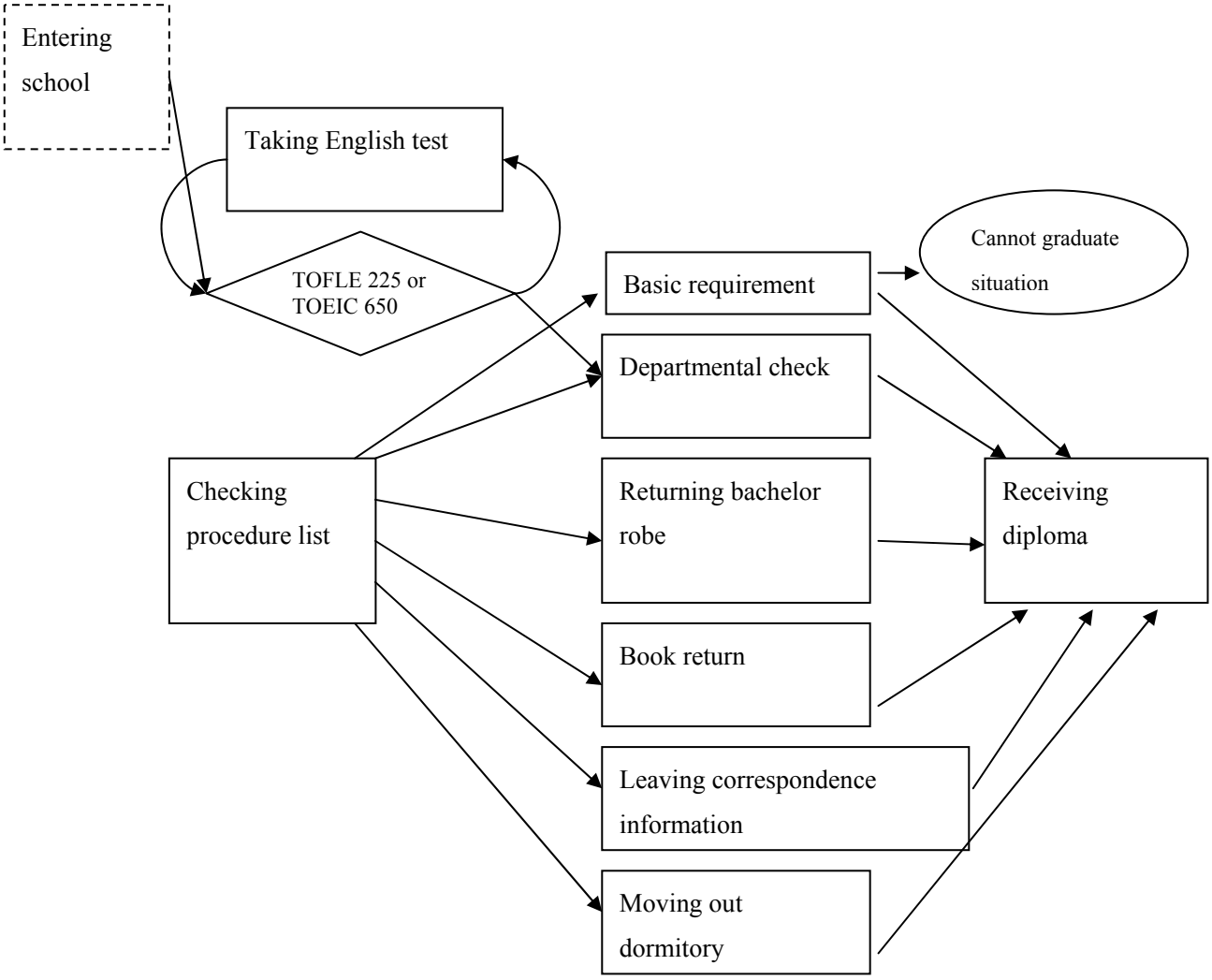

Figure 3-2: Task flow of student leaving school process

Table 3-1: Tasks in a student leaving school process

id	Loc_time	participant
Checking procedure list	<internet,∅>	N/A
Credit check	<Dean's office, hour 08:00~17:00>	Office of academic affairs
Departmental check	< Office of faculty, hour 08:00~17:00>	Office of faculty

Returning bachelor robe	< Office of general affairs, hour 08:00~17:00>	Office of general affairs
Book return	< Library, hour 08:00~17:00>	Library
Leaving correspondence information	<office of graduate assistance, hour 08:00~17:00>	Office of student affairs
Moving out dormitory	<service counter of dorm, hour 08:00~17:00>	Office of student affairs
Taking English test	< examination room, hour 08:00~12:00>	Center of English test
Receiving diploma	<Office of academic affairs, hour 08:00~17:00>	Office of academic affairs

Table 3-2: Data dependencies of the graduation process

task	input	output
Checking procedure list	∅	Checklist
Credit check	{Student ID, Checklist}	Checklist
		Checklist
Departmental check	Passed English score sheet, Checklist	Passed English score sheet, Checklist
Returning bachelor robe	Bachelor degree clothes, Checklist	Checklist

Book return	Books, Checklist	Checklist
Leaving correspondence information	Checklist	Checklist
Moving out dormitory	Checklist, Dorm key	Checklist
Taking English test	∅	Passed English score sheet
		∅
Receiving diploma	Checklist	diploma

Chapter 4 Verifying Personal Processes

4.1 Mapping a personal process onto Petri Nets

In the underlying process model, we use timed Petri Nets because time is an important factor in personal processes. A time interval as specified by the two time parameters, [T_{1j}, T_{2j}], is associated with a transaction t_j, to confine the effective period in which t_j can be fired. The first element, T_{1j}, is the earliest firing time (EFT) of the transition, and the second one, T_{2j}, is the last firing time (LFT) of the transition. When a transition is ready to fire, it won't be able to do so before reaching its EFT, and it must fire before reaching its LFT. As shown in figure 4-1,each transition may be associated with a specified time interval that limits its executable period.

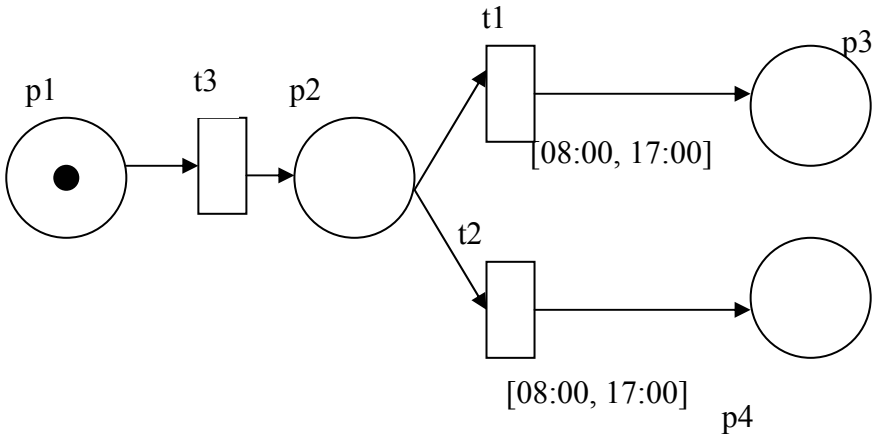

Figure 4-1: An example timed Petri Net

- Mapping personal tasks

We map each personal task as a transition. The EFT and LFT of a transaction is defined as the minimum and maximum of the executable times of the pertaining task respectively..

- Mapping data items

We map each data item as a place.

- Mapping input data of tasks

There is an arc linked from a data item *d* to a transition *t* if *d* is an input data item of the task represented by *t*.

- Mapping output data of tasks

For a task *t* with more than one thread, a place *c* and a link (*t*, *c*) are constructed. In addition, for each thread t_i of *t*, a transition is generated with an input from *c*, and an arc from the transition of t_i to the place of each output data item is created. For example, Figure 4-2 shows the corresponding Petri Nets for modeling a task *t* with three threads h1, h2, and h3, where the output data items are (a, b), (c), (a, d) respectively.

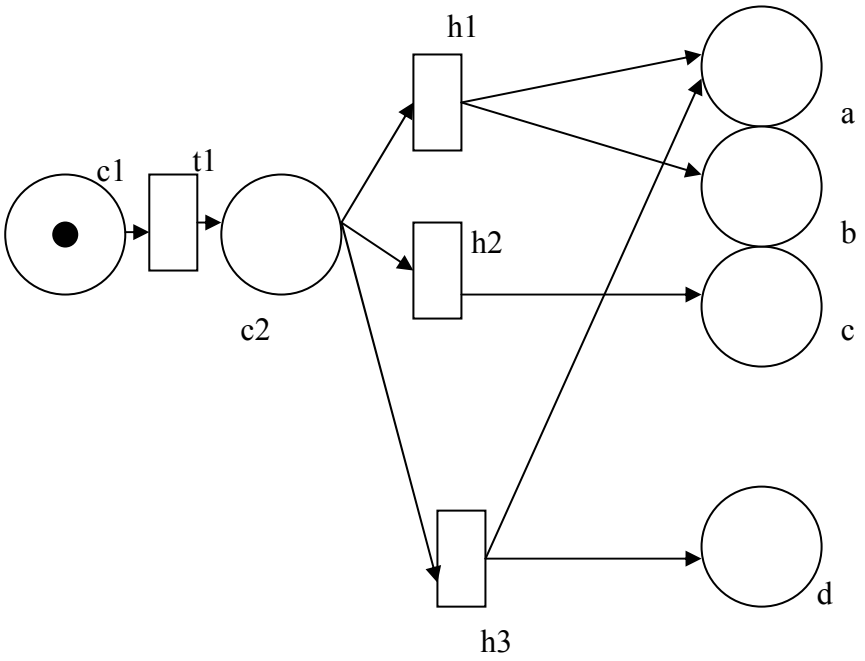

Figure 4-2: The Petri Nets of a task with three threads

- Mapping Goal

A goal represents a desired execution result of a personal process. Thus, a goal can be denoted as a set of places called target set, where each place could represent a data item or completion of a particular task.

- Mapping the available of data items

We put a token in a place to represent the availability of the corresponding data item.

- Mapping control flow

The control flow of personal process can be gracefully specified in the way as described in the literature review. Note that extra places may be added in the Petri Net for realizing some control constructs. The control flow is enforced by the flow of the tokens and the routing constructs.

The Petri Net of the example student graduation process

In the example student graduation process, there are nine tasks each of the nine tasks

and each of their threads is modeled as a transition. The first task in the process is “Checking procedure list”, and the task needs student to check his process on internet. We would model the task as a transition t1, and the next is a parallel sequence and we would model it by and-split. In order to connect the threads and task, we need to model it with condition flow. As we can see in the following figure, t2 is a task and h3 and h4 is its threads, and the connection of main task to its thread is c8. The output data of the two threads is d19 and d20. It is the same in t10 task, but one of its threads would iterate to c8. The other task does not contain thread, so we can model it strait by one transition and its output data and its control place. The final task “Receiving diploma” is modeled as t13 and it will lead to our final goal d29. It is an and-join and it needs other task had finished. The description of transitions and some resource data is in table 4-1 and 4-2.

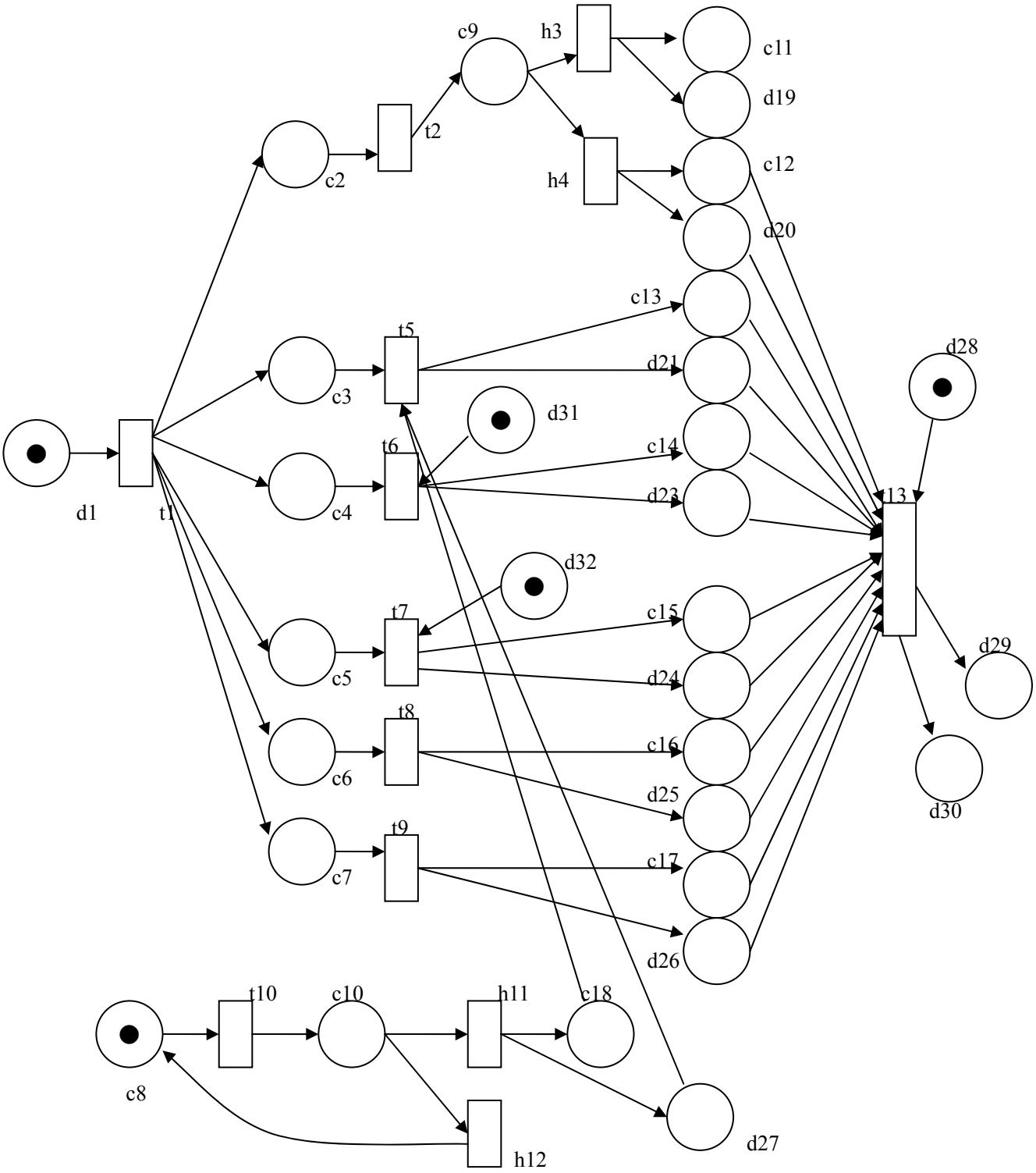

Figure 4-3: Student leaving school process with Petri Net

Table 4-1: Descriptions of Transitions in Figure 4-3

Item	Task name

t1	Checking procedure list
t2	Credit check
t5	Departmental check
t6	Returning bachelor robe
t7	Book return
t8	Leaving correspondence information
t9	Moving out dormitory
t10	Taking English test
t13	Receiving diploma

Table 4-2: Descriptions of resource places in Figure 4-3

d1	Checklist to leave school
d19	Not passed qualification
d20	Passed qualification
d20~d26	Stamp for each task
d27	Passed score sheet
d28	two inch picture
d29	Diploma
d30	graduate stamp
d31	Bachelor degree clothes
d32	Library books

4.2 Constraints of the personal workflow

There are a number of constraints associated with personal processes. In a personal process, tasks are associated by their respective attribute values, which may implicitly determine their execution order, in addition to the order explicitly determined by

control flows. For example, if a task T_2 requires a data item that can only be produced by T_1, T_2 will not begin executing before T_1 terminates. We use the binary relation $\prec$ to represent the temporal relation of the two events.

Definition 4.1 (Process-aliveness)**:** A personal process is said to be alive (or preserve the process-aliveness constraint) if the target set P_t is reachable from the source data set P_s.

The process-aliveness constraint ensures that there is a chance that all the desired data will become available. When the process-aliveness constraint is violated, it makes no sense to continue executing the process.

Definition 4.2: An unexecuted task t in a personal graph P is said to potentially contribute to a place d if there exists an executable simple path of P to d that starts with some thread in t.

Definition 4.3 (Task-aliveness)**:** An unexecuted task t is said to be alive if

1. t has not expired,
2. All the input places of t are reachable from P_s, and
3. t potentially contributes to at least one place in P_t.

When a task is alive it is possible to be executed, and its execution may contribute to the availability of at least one place in the target data set. Furthermore, if a personal process does not preserve task aliveness, some tasks become the garbage in the process, and it needs to be cleared.

Consider a personal process shown in Figure 4-4 that is revised from the one shown in Figure 4-3, where task “Taking English test” does not connect to task

"receiving diploma". It is therefore meaningless for the user to execute the task because such a task does not contribute to the receiving of the diploma. In this case, "Taking English test" is not alive and can be deleted.

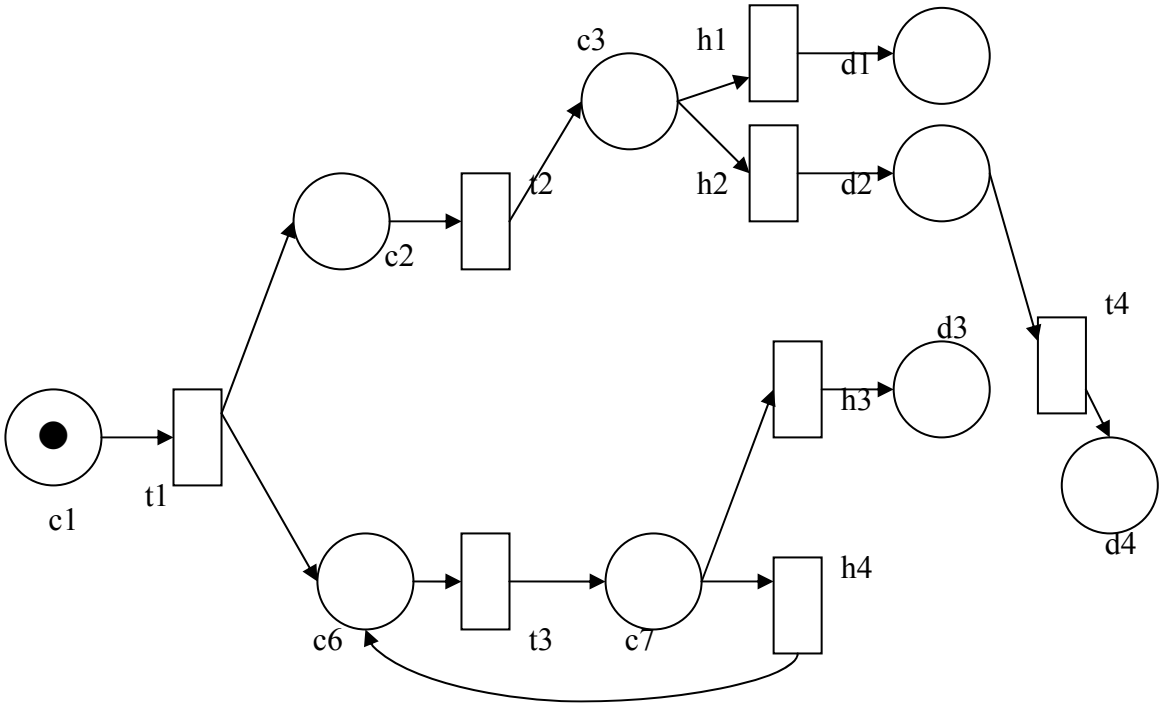

Figure 4-4: Revised Student leaving school process

Definition 4.4: (Correctness) A personal process (instance) is said to be correct if it is alive, and all its unexecuted tasks are alive.

- A personal process can be incorrect for the following reasons:
    1. Some tasks are not alive during a normal process execution (and should be removed).
    2. The user may have made some mistakes in designing or changing the personal process.
    3. The personal process was so badly executed that it makes no sense to continue executing the remaining unexecuted tasks.

The task aliveness and the process aliveness of personal processes can be verified by looking at the reachability tree of the corresponding Petri Net with some modification.

In Chapter 4, we describe how to maintain the reachability tree and verify the correctness of personal processes.

Chapter 5 Analyzing the constraints of personal workflow

This chapter presents our approach to verifying the correctness of a personal process. To do so, we first construct the reachability tree of the personal process (modeled in Petri Net) by using the algorithm described in [Mura89]. The algorithm is re-iterated in the appendix for self-containment. The correctness of a personal process can be subsequently verified by looking at its reachability tree. Figure 5-1 shows part of the reachability tree of the Petri Net depicted in Figure 4-3.

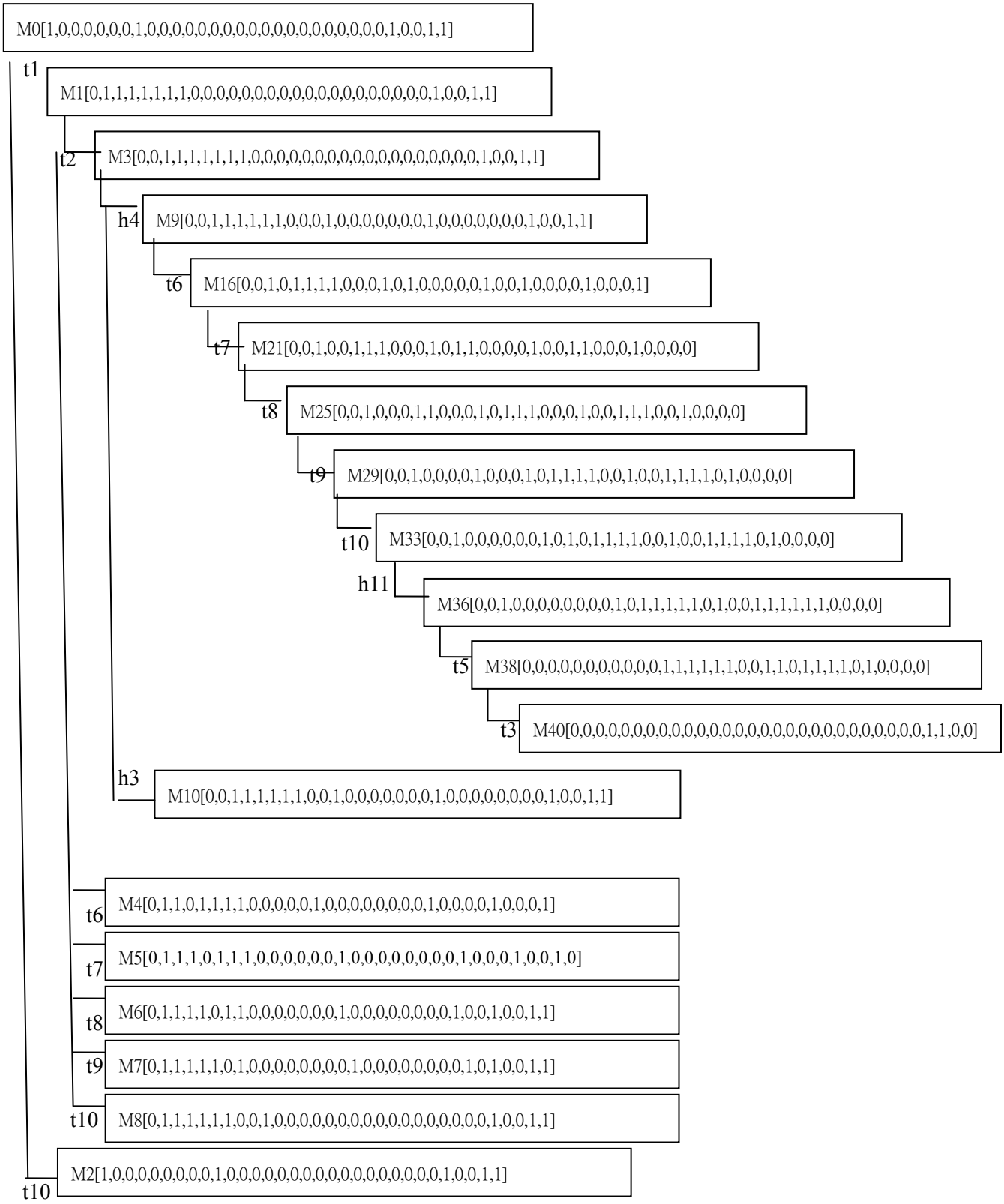

Figure 5-1: Part of reachability tree of the Petri Net shown in Figure 4-3

5.1 Process-aliveness

A personal process is said to be alive (or preserve the process-aliveness constraint) if the target set is reachable from the source data set. This can be easily verified by checking if any marking that covers all the target places (i.e., each target place has a token in it) is reachable from the source data set. Take the student graduation process as an example. The initial making is such that each of d1, c8, d28, d31, d32 has a token in it, and the target set contains only a place d29. The verification of the process-aliveness of a personal process is equivalent to checking whether there exists a node *n* in the reachability tree of the personal process such that the marking of *n* covers the target marking and there exists a path from the current node to *n*. In figure 5-1 we can find that the path (t1, t2, t4, t6, t7, t8, t9, t10, t11, t5, t13) that leads M0 to M40, which has one on the target data item d_{29}. Thus, we conclude that the personal process is alive. On the other hand, if the process executes t3, the current node becomes M10. In this case, the personal process is no longer alive because it will never reach any node with token in the place of d_{29} from M10. Thus, the entire personal process does not need to be further executed.

5.2 Task-aliveness constraint

Recall that an unexecuted task is said to be alive if

i. has not expired,

ii. The marking comprised by the input data is coverable.

iii. There exists a path *P* that reach target from source and *P* contains *t*

To verify task-aliveness of a task *t*, we first identify a set of nodes who markings

cover the target set, called *target nodes*. We then discover all paths that leads from the current node to any node in the target nodes. If *t* appears in any of the paths, we can conclude that *t* is alive.

5.3 Algorithm for verification

The algorithm starts with the construction of the reachability tree. It then enters a loop and waits for events of the following types: TASK_START, TASK_COMPLETE, and TASK_EXPIRED. When a task starts, the state of the task is changed to “executing”. When a task complete, the sequence of firing transitions, as returned by gettask_transitions( ), is used to propagate the reachability tree by calling UpdateTree( ). The resultant tree is then checked for process aliveness by calling Process-Alive( ), followed the check for task aliveness. Note that to check task aliveness, all the paths that lead from the root to every node that covers the target marking are identified to form the new reachability tree by calling SearchTargetUpdateTree( ). Tasks that do not show up in the reachability tree are pronounced as *dead* tasks. When a task expires, the associated transitions are deleted from the reachability tree. Then the same procedure for checking process aliveness and task aliveness are performed.

Input P: a personal process;
M: a target set;
Procedure PersonalProcess-Execution:
01 Construct a reachability tree *T* of *P*;
02 Process-Completion = FALSE:
03 while(!Process-Completion){
04 GetEvent(*e*);
05 Switch(*e*){
06 case *e*.type='TASK_START':

```
07              t.state=EXECUTING;
08              break;
09          case e.type='TASK_COMPLETE':
10              t.state=COMPLETE;
11              UpdateTree(Tree T, e.gettask_transitions(), “COMPLETED”);
12              if T.root covers M {
13                  Process-Completion = TRUE;
14                  break;
15              }
16              If ( ! Process-Alive(Tree T, mark Mark_target))
17                  Notify(“Cannot reach the target”);
18              SearchTargetUpdateTree (Tree T, mark Mark_target);
19              for each unexecuted task t do
20                  if t does not appear in T do
21                      t.state=DEAD;
22              break;
23          case e.type=”TASK_EXPIRED”
24              UpdateTree(Tree T, e.gettask_transitions(), “EXPIRED”)
25              If ( ! Process-Alive(Tree T, mark Mark_target))
26                  Notify(“Cannot reach the target”);
27              SearchTargetUpdateTree (T, M);
28              for each unexecuted task t do
29                  if t does not appear in T do
30                      t.state=DEAD;
31              break;
32          }}
```

Figure 5-2: Pseudo-code for dynamically verifying a personal process

The Process-Alive function is based on BFS(breadth first search) to search the reachability tree once it finds one marking covers the target marking it would stop searching. Take the student leave school process for example, the function would first search M_0 in Figure 5-1 and the following search sequence is M_0, M_1, M_2, M_3 …. The searching will continue until one marking covered the target marking or no more marking to search.

Input: *T*: the coverabaility tree of the process.
Mark_target: the target marking for the user.
Output: Boolean factor to present 'Process-alive' or 'not process-alive'
Procedure Process-Alive:

```
01  queue que;
02  node start = T.getroot();
03  if(Mark_target ⊄ start){
04        start.status = 'traversed';
05        insert(que, start);
06        while(empty(que) = FALSE ){
07              x = remove(que);
08              for each neighbor nd of x do{
09                    if(nd.status ≠ 'traversed') {
10                          if(Mark_target ⊄ nd){
11                                nd.status = 'traversed';
12                                insert(que,nd);
13              }else
14                    return true;
15        }}}
16        return false;
17  }else
18        return true;}
```

Figure 5-3: Pseudo-code of process-alive

In Figure5-4, the SearchAllTarget function will use the breadth first search to find all nodes that cover the target marking and all the dead tasks. The function will first trace the whole tree and find all marking that covered the target marking. For these markings the next step was to back track its transition sequence and store into liveset. For all transitions deduct the liveset the rest was dead transition. For one task there may contain some threads and some may reach the target data some may not and if one thread from the task can reach the target data then the task is alive. For the reason the third step is to track all task if it's all threads are dead then the task is dead, too.

The task" Examine qualifications to graduate" in Figure 4-3 contains three transitions t2, h3, and h4. Although h3 can not reach the final target data p29, the other thread t4 can reach it and the task matches task-aliveness constraint.

Input: *T*: the coverabaility tree of the process.
Mark_target: the target marking for the user.
Output: *deadtaskset*: one list that contains all dead tasks.
Procedure SearchAllTarget:

```
01  queue que;
02  list liveset;
03  list deadset;
04  list deadtaskset;
05  node start = T.getroot();
06  start.status = 'traversed';
07  insert(que, start);
08  while(empty(que) = FALSE ){
09      a = remove(que);
10      x = GetFirstChild(a);
11      for each neighbor nd of x
12          if(Mark_target ⊂ nd)
13              liveset.add(nd.previoustask);
14          else
15              insert(que,nd);
16  }
17  for each node n in tree T
18      if (node not in liveset)
19          deadset.add(n);
20      for each task ta do {
21          is_dead = TRUE;
22          for each transition tr in ta
23              if(tr.mark= "task_end")
24                  if(tr not in deadset)
25                      is_dead = FALSE;}
26  if(is_dead = TRUE)
27      deadtaskset.add(ta);
28  return deadtaskset;}
```

Figure 5-4: Pseudo-code of Task-alive

- **Update the reachability tree**

A personal process may be correct at design time but become incorrect at execution time. Therefore, the reachability tree may change, and in order to make sure the reachability tree is correct, we need to update the tree for following situation.

- Task expired and deleted：When user deleted one task or it is expired, we can delete the sub tree of the task.
- Task Complete：From time goes by user will complete some task, in order to maintain one efficient tree we need to adjust the tree. In reachability tree if one task completed we will move the pointer and keep the root node be the current marking. Because the leaf node may be “old” and we move the pointer may lead the “old” node loses information. So if one task completed we will track the leaf node if it refer to the root node we will need to copy the whole sub tree. If the current marking will not reach the target marking we will notify user that need not to continue.

The algorism is as following:

Input: *T*: the coverabaility tree of the process.
status: the message of different condition.
updatetask: the task that has been changed.
Output: *deadtaskset* that contains all dead tasks.
Procedure UpdateTree
01 if(*status* = ”EXPIRED”)
02 deletesubtree(*updatetask*);
03 if(*status* = “DELETED”)
04 adjusttree(*updatetask*)
05 elseif(*status* = ”COMPLETED”){
06 for each child node *cn* of *T*.root
07 if (*nd*.previoustask = *updatetask*){
08 for each leaf node *ln* in *T*

```
09                      if(ln.status = ”old”)
10                          referednode = findrefernode(ln);
11                              if(referednode = T.root)
12                                  copysubtree(ln, refernode);
13                      T.root = nd;
14          }}}
```

Figure 5-5: Pseudo-code of UpdateTree

Take the student leave school process for example, if user completed t1 transition then the root would be M_1 in figure 5-1. The user may next execute t2 transition and the root would also transfer to M_3. For the change of the root, some leaf node may refer the root node and if we delete the root node it may lose the reference. For example if user had executed to M_{10} in figure 5-1 and now the root is M_{10}. The user will execute other transition next and the leaf node M_{12} referring to M_{10}. In the situation we would copy the sub tree of M_{10} to M_{12}. Then the reference would not be lost.

If “Take exam” task was expired, the update of the tree would delete the sub tree of the three transitions t10, h11, and h12.

If one task is dead there may still leave task start transition on the reachability tree. One task may contain more than one transition in our modeling process. One transition in task would contain one label to distinct as a task start transition or a task end transition. If one task is dead we would need to clear the transitions on the Petri Net and the reachability tree. It is because the rest of the transition may lead user to dead way and the information is also usefulness to store. The method to clear one dead task is the same as one task is expired.

Chapter 6 The implementation

This chapter describes our experience in implementing a prototype PWFS. The PWFS system supports the design of a personal process, the execution of a personal task, and the queries to the personal process. To do so, the architecture involves four components: the template providers, the service providers, the online execution engine, and the PWFMSs. The functions of each component are described below:

- **The Template Provider** provides a set of personal process templates; each suited to a user with distinct background and needs for achieving a personal goal. After selecting a particular template, a user can use his handhold device with a PWFMS installed to download the template.
- **The Service Provider** tracks the execution status of a task (possibly through an enterprise workflow management system) and publishes a set of Web services that reveal tasks' execution status and available (*partial*) personal processes.
- **The Online Execution engine** allows the user to update the execution status of a personal process and to perform correctness verification. Specifically, upon receiving an update, it checks all the constraints and updates the reachilibity tree.
- **The PWFMS** is executed on handhold devices to manage personal processes. It is capable of downloading personal processes from template providers and interacts with service providers for keeping track the execution status of tasks. In addition, it also provides interfaces for a user to change task execution status and place queries. However, it does not check the correctness of a personal process for each update due to the limited computing and storage capacility. The request for correctness verification, upon the request of the user, is sent to the online execution engine, which has the capacity for correctness verification.

The relationship between the four types of components is depicted in Figure 6-1. The interactions between different types of components are conducted via Web services. For the interaction among service providers, template provider, and PWFMS, please see [Tu03] for details. Our focus in this work is on using Petri Net for executing and verifying personal processes. Therefore, in the following, we will describe the design of online execution engine and PWFMS in detail.

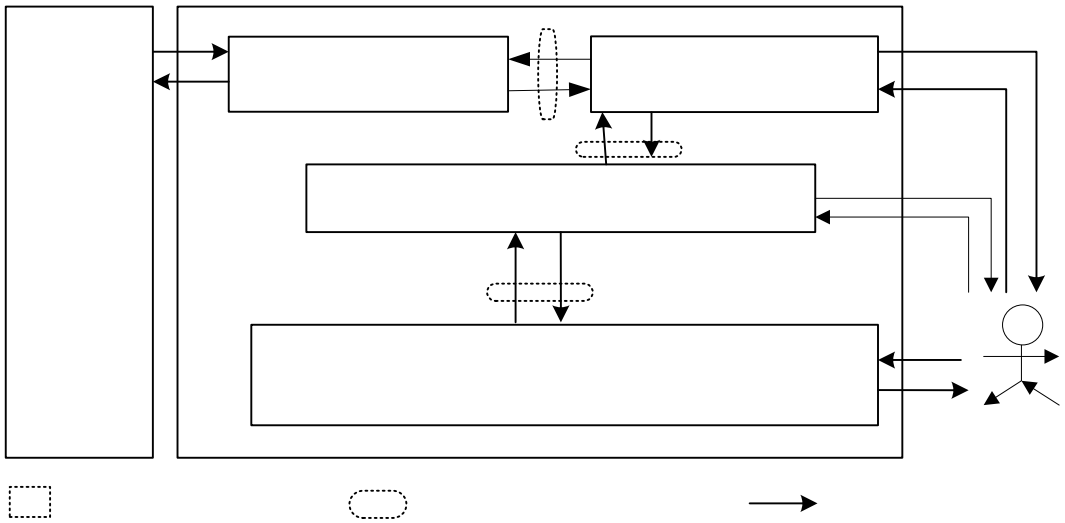

Figure 6-1: Logical architecture of the entire system

The service provider, template provider and the online executing provider run on servers located on the fixed network. All Web services involved in the three components are implemented by Perl and execute on Apache Web server. The template provider is a Web-based system using Apache as the Web server and MySQL as the underlying DBMS. We use PHP for implementing the Web interface and the Web services. The PWFMS executes on hand-hold devices and is implemented by using J2ME-MIDP (Java 2 Mobile Information Device Profile) as the development tool.

6.1 Online execution engine

The online execution engine allows the user to update and track the status of a personal process on the Internet. Specifically, it has two objectives: to support the update of personal process status and to ensure the satisfaction of all the constraints on a personal process. To satisfy both goals, the online execution engine has the following modules: the *execution manager*, the *constraint verifier*, and the *UI Manager.* The executing manager is the core module of the Online executing provider, and when receiving status update from UI Manager, updates the reachability tree and verify the correctness of the resulting personal process. In addition, the executing manager provides two web services for verifying process aliveness and task aliveness given a sequence of completed threads. They are intended to be invoked by PWFMS, as described below, for correctness verification. The two published Web services, namely process_alive and task_alive, are of the following formats.

- Process_alive:
  - Request: requires “user identifier” and “personal process name” and “executed tasks”

```
<request>
    <user>name1</user>
    <process> process1</process>
    <path>
        <task>t1</task>
        <thread>h1</thread>
        <task>t1</task>
        <thread>h1</thread>
    </path>
```

```
</request>
```

- Response: true or false to represent meet the constraint or not

```
<response>
    <answer>correct</answer>
</response>
```

– task_alive:

- Request: requires “user identifier” and “personal process name” and “executed tasks”

```
<request>
    <user>name1</user>
    <process> process1</process>
    <path>
        <task>t1</task>
        <thread>h1</thread>
        <task>t1</task>
        <thread>h1</thread>
     </path>
</request>
```

- Response: a set of dead tasks

```
<response>
    <task>task1</task>
    <task>task2</task>
</response>
```

Online execution engine runs on a server located on the fixed network. Figure 6-6 is a screenshot of an executed process. To manipulate tasks of a personal process Online executing provider includes functions as follows: task status change, data status change. Task status change would change the task status as mention in chapter 3. Figure 6-7 shows screenshot of constraint check after task executed. Data status change would change data to available and unavailable. It also checks constraints and updates reachibality tree as mentioned in chapter 5.

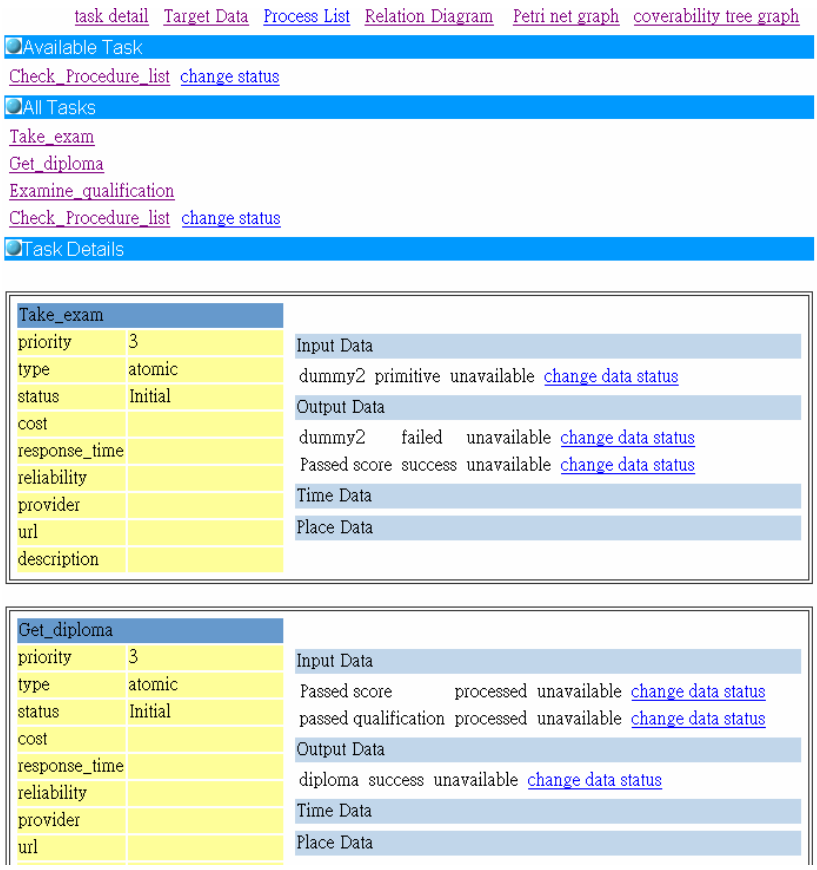

Figure 6-6: Task detail of a personal process

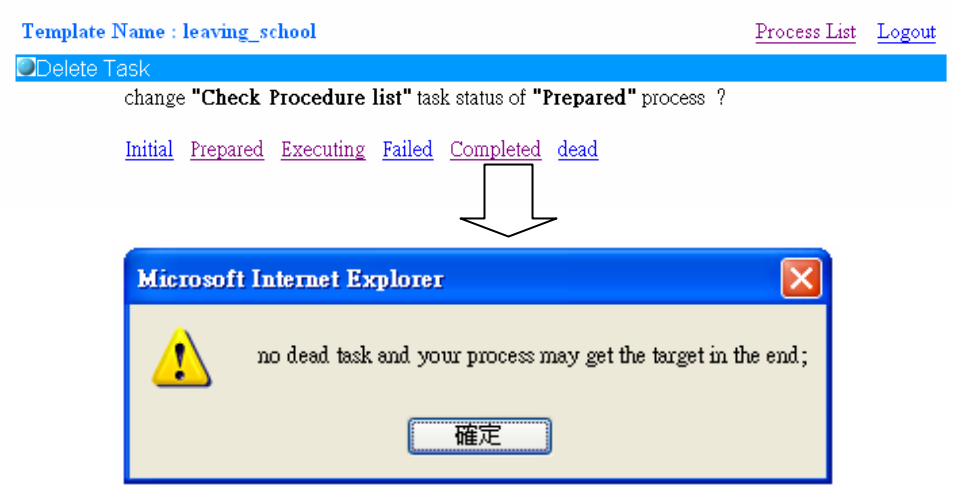

Figure 6-7: Constraint check

6.2 Personal workflow management system

The PWFMS enables users to create or fetch a new personal process (from the template provider), to manage existing personal processes, to interact with the service providers of organizations that are responsible for executing tasks. The *WS agent* module is responsible for interacting with the template provider (e.g., for retrieving and downloading personal processes by issuing listTemplate and getTemplate Web services respectively) and with the service provider (e.g., for getting the execution status of a task by issuing getTaskStatus Web service). For detailed description about the architecture and the functions of the PWFMS, please refer to [Tu03]. Our focus in this work is on the verification of personal process correctness. Thus, we will describe the execution of a personal process. Specifically, the PWFMS perform the following steps for a personal process:

1. It downloads the process definition, including its underlying Petri nets, from the template provider located on the Internet.
2. Upon the request of the user, the PWFMS changes the status of

activities by modifying the current marking on the Petri net.

3. When the Internet is available, the PWFMS synchronizes with the service execution engine by uploading the status change of the personal process since the last synchronization. The service execution engine updates its status, perform correctness checking, and inform the PWFMS whether the resulting personal process status is correct.

Note that the PWFMS does not perform online correctness checking at Step 2. Instead it batches a sequence of status updates and sends them to the service execution engine periodically for correctness checking, as shown in Step 3. This design relieved the PWFMS from the tremendous overhead of maintaining and traversing reachability tree.

Figure 6-8: Petri Nets data list in PDA

Figure 6-9: Downloading the process definition from the template provider

Figure 6-10: Change task status function

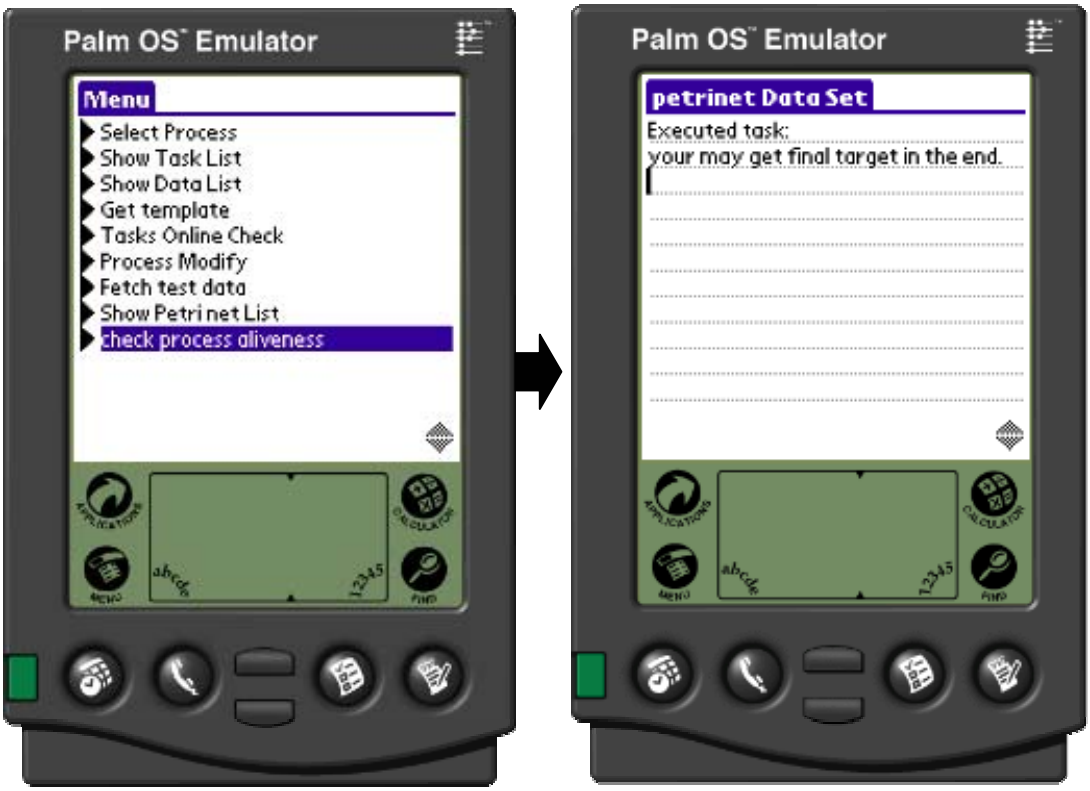

Figure 6-11: Constraints check on PDA

Our implementation efforts lead to the following lessons:

1. The current support for web service communication on handheld devices is limited.
2. To model a process, the interface and easy-to-use is important.
3. Recall that we maintain a reachability tree for verifying the process correctness. As expected, the number of nodes in the reachability tree dramatically increases with the increase on number of tasks. In our example personal process, it takes only about three seconds to build a reachability tree, which is acceptable. However, for a larger personal process, reachability tree could be huge and takes longer time to traverse. We have conducted an experiment by adding more tasks in the parallel construct in our example personal process. Figure 6-12 shows the storage requirement of reachability tree with respect to the increase on the number

of tasks in the parallel construct. It can be seen that the storage space dramatically increases when the number of tasks is more than 7. Besides, Figure 6-13 shows the time required to build a reachability tree. Again, the time increases dramatically when the number of tasks in the parallel construct is more than 7. Figure 6-14 shows the time required to traverse a reachability tree at the request for the correctness checking at the beginning of a process instance. It can be seen that the tree traversal time moderately increases with the increase on the number of tasks, though it still takes more than 2 seconds under various settings.

The lesson we learned from the third point also justifies our design decision in maintaining the reachability tree and conducting correctness checking only on the online execution engine, rather than on the PWFMS, due to the limited storage and computing capacity associated with most current handheld devices. However, should the computing power and storage of hand held devices increases in the future, it would make sense to implement the correctness checking on the PWFMS.

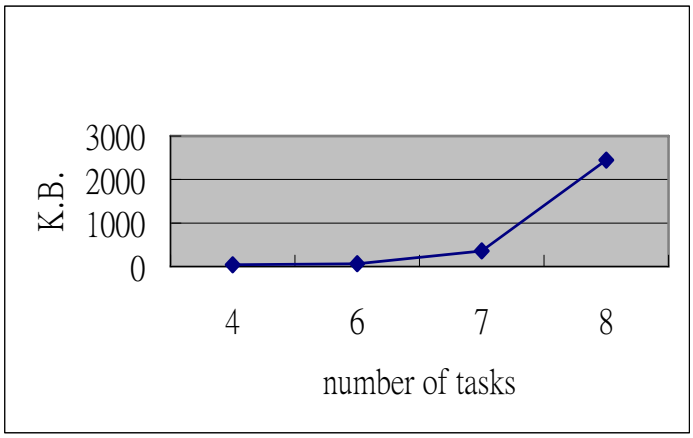

Figure 6-12: The storage requirement of the reachability tree

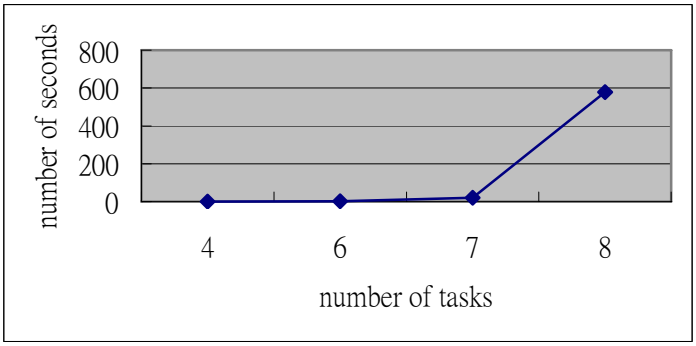

Figure 6-13: The time requirement for building the reachability tree

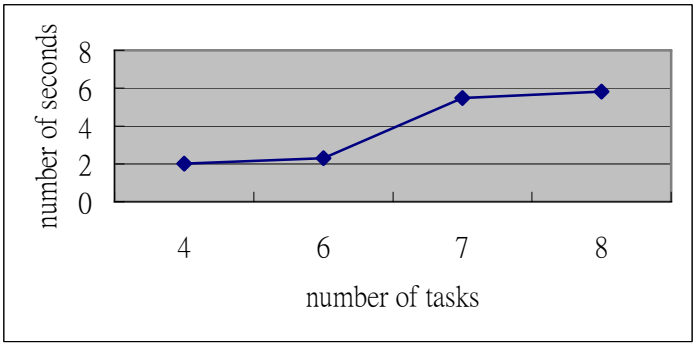

Figure 6-14: The time requirement for traversing the reachability tree

Chapter 7 Conclusions

In this thesis, we have modeled a personal process using Petri Nets to describe both the control flow and data flow pertaining to the personal process. We also defined the correctness of a personal process and proposed the verification approach based on the Petri Nets of the personal process. A prototype has been constructed to show the feasibility of our approach. In the prototype, the correction verification approach is implemented on the online execution engine located on the fixed network. The PWFMS, which executes on some handheld device, tracks the execution status of a personal process and send the status change in batch to the online execution engine for correctness check once the PWFMS is connected onto the Internet. In the future, we plan to apply the concepts and implementations of personal workflow management to more specific domains.

Appendix

Algorithm for building the reachability tree [Mura89]

Step 1) Label the initial marking M_0 as the root and tag it " new".

Step 2) While "new" marking exist, do the following:

Step 2.1) Select a new marking *M*.

Step 2.2) If *M* is identical to a marking on the path from the root to *M*, then tag *M* "old" and go to another new marking.

Step 2.3) if no transitions are enabled at *M*, tag *M* "dead-end".

Step 2.4) while there exist enabled transitions at M, do the following for each enabled transition *t* at *M*:

Step 2.4.1) Obtain the marking *M'* that results from firing *t* at *M*.

Step 2.4.2) On the path from root to *M* if there exists a marking *M"* such that $M'(p) \geq M"(p)$ for each place *p* and $M' \neq M"$,ie., *M"* is coverable. Then replace $M'(p)$ by ω for each place *p* such that $M'(p) > M"(p)$.

Step 2.4.3) introduce *M'* as a node, draw an arc with label *t* from *M* to *M',* and tag *M'* "new".

References

[Aals03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kie-puszewski, and A.P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, 14(3), 2003, pp.5–51.

[Aals94] W.M.P. van der Aalst, "Putting Petri nets to work in industry," *Computers in Industry*, 25(1), 1994, pp.45–54.

[Aals98] W.M.P. van der Aalst,"The Application of Petri Nets to Workflow Management," *The Journal of Circuits, Systems and Computers*, 8(1), 1998, pp.21-66.

[Adam98] N.R. Adam,V. Atluri, W.-K Huang, "Modeling and Analysis of Workflows Using Petri-Nets," *Journal of Intelligent Information Systems*, 10(2),1998.

[Chen01] Y.-F. Chen, "The Research on Personal Workflow Systems in Support of Pervasive Computing," *Master thesis, National Sun Yat-sen University,* July 2001.

[Daya91] U. Dayal, M. Hsu, and R. Ladin, "A Transactional Model for Long-Running Activities," Proceedings of the 17th International Conference on Very Large Data Bases**,** 1991.

[Gepp98] A. Gppert, D. Tombros, and K. R. Dittrich, " Defining the Semantics of Reactive Components in Event-Driven Workflow Execution with Event Histories", *Information Systems*,23(3), 1998.

[Hee94] K.M. van Hee,"Information System Engineering: a Formal Approach,"Cambridge University Press, 1994.

[Hwan03] S.-Y. Hwang, Y.-F. Chen, "Personal Processes: Modeling and Management," *4'th Int'l. Conf. on Mobile Data Management (MDM03), Melbourne, Australia*, Jan. 2003. LNCS2574, Springer Verlag, pp.141-152.

[Kao04] P Kao, "Constraints and QoS Management of Personal Process," *Master thesis, National Sun Yat-sen University,* July 2004.

[Kiep00] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler, "On Structured Workflow Modelling," *Proc. of the 12th Int. Conference on Advanced Information Systems Engineering* (CAiSE00), 2000.

[Jens96] K. Jensen. "*Coloured Petri Nets. Basic concepts, analysis methods and practical use,"* EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1996.

[Lin02] M. J. Lin, "A Framework for Providing and Executing Workflow Templates in a Mobile Environment," *Master thesis*, *Information Management Department*, *National Sun Yat-sen University*, July 2002.

[Nidd94] M. Nidd, "Time Extensions of Petri Nets," 1994., available at http://citeseer.nj.nec.com/nidd94time.html

[Mura89] Murata, T., "Petri Nets: Properties, Analysis and Applications," *IEEE Proceedings* ,77(4), April 1989, pp. 541-580.

[Petr62] C.A. Petri. "*Kommunikation mit Automaten*," *PhD thesis, Institut fur Instrumentelle Mathematik*, Bonn, 1962.

[Tu03] J.K. Tu, "Personal Workflow Systems in Support of Inter-process Integration," *Master thesis, Information Management Department*, *National Sun Yat-sen University y,* July 2003.

[WFMC96] WFMC. WorkflowManagement Coalition Terminology and Glossary (WFMC-TC-1011). Technical report, Workflow Management Coalition,

Brussels, 1996.

[Wodt97] D. Wodtke, and G. Weikum, “A Formal Foundation for Distributed Workflow Execution based on State Charts,” *Proc, of the Int. Conf. on Database Theory*, Springer LNCS 1186, 1997.