Modeling and Analyzing Timed Changes within Workflow Systems

Clarence Ellis
Karim Keddara
Jacques Wainer

CU-CS-869-98

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Modeling and Analyzing Timed Changes within Workflow Systems

Clarence Ellis[1] and Karim Keddara[1] and Jacques Wainer[2]

[1] University of Colorado, CTRG Labs, Dept of Computer Science, Boulder CO 80309-0430, USA
[2] University of Campinas, Dept of Computer Science, Campinas Brazil

Abstract. Workflow Management Systems[26] are networked computer systems which enable the specification, analysis, coordination, and enactment of organizational procedures. Although some of these workflow management systems (we abbreviate to workflow systems) have been successful, many have failed to improve organizational processes. One reason for this failure is the dynamically changing nature of organizations and work which is not well supported by workflow systems.
In a previous paper[9], the authors defined notions of dynamic change in workflow systems by utilizing Petri net models. Some types of workflow change are safe, non-disruptive, and can be performed anytime. Other changes disturb ongoing transactions, and cause problems if they are attempted in a dynamic fashion. That previous paper also presented various definitions of dynamic change correctness.
In this paper, we define the *timed flow nets* as a way to accommodate time issues into the design of workflow systems and the analysis of their static structural changes. We also expand upon the issue of "safe" structural transformations which preserve the soundness property[20]. This paper introduces another new Petri net based model, the *Timed Adaptive Flow Nets* model, that is suitable to address dynamic structural timed changes within workflow systems. This model generalizes the notions of SCOC[9] and Extended SCOC[13].
This work is part of an ongoing research effort of the Collaboration Technology Research Group (CTRG) at the University of Colorado. Previous CTRG work introduced the many dimensions of workflow that can change, including process change, change of roles and actors, application data change, organizational structure change, and change of social structures. In this paper, we focus on process changes.

Table of Contents

List of Figures

1 Modeling Workflow Procedures

We assume the reader to have a basic understanding of the Petri net models, their firing semantics and some of their basic properties including boundedness, safeness, liveness (the reader is referred to [17, 15] otherwise.)

Many Petri-net based workflow models have been introduced in the literature[7, 8], but only few of them deal with time issues. Meeting commitments and deadlines have been identified as a key requirement in organization business models to achieve customer retention and expansion. Therefore, there is an urgent need to accommodate the temporal behavior of workflow systems. This situation has been acknowledged but not addressed by the workflow management coalition[26]. On the other hand, many efforts have and are being put in place to address the issue in many other areas; including real time systems, communication protocols, process planning, work-force management systems etc...

In a previous work[9] workflow procedures are modeled by the so-called flow nets, this modeling is carried out as follows: activities that define the procedure are represented by transitions. Each transition has a label, a set of input places to mark the beginning of the modeled activity, and a set of output places to mark the end of the activity. The workflow procedure also specifies the order in which its activities ought to be carried out; activities may be mandatory or optional, they may be executed in sequence or in parallel. This partial ordering is modeled in the net by the so-called flow relation. Each flow net has a single entry place to reflect the start of the modeled procedure and a single exit place to mark the end of the modeled procedure. Furthermore, each element of the flow net is in a path which links the entry place to the exit place.

1.1 Timed Flow Nets

Different ways of accommodating time in Petri net models have been proposed by many researchers. These different proposals were influenced by the specific application domains, however there seem to be a commonly shared concern not to modify the basic behavior of the untimed model (parallelism and non determinism.) These proposals may vary in their choice of the components (i.e. places or transitions or flow) to be timed. For instance, timing is associated with places in [18], with transitions in [16, 14, 6, 27, 10, 2, 3], and with the flow relation in [22, 5]. They may also differ on how timing information is modeled. It may be fixed as in [16, 18], a time interval as in [14, 27, 18, 22, 5], randomly distributed as in [2, 3], or arbitrarily continuous as in [10].
Without claiming the superiority of any particular proposal with respect to others, we adopt a timed transition approach which uses time intervals. Our choice is pragmatic and is driven by our concern to use a model which is in the middle of the complexity spectrum. It is well known that the modeling power of timed place Petri nets is equivalent to the limited modeling power of timed transition Petri nets with fixed durations. Although the analysis of Stochastic Petri Nets is possible under certain somewhat severe conditions, the behavior of these nets is better analyzed under simulation. This does not mean that timed transition Petri nets do not resist any kind of analysis. On the contrary, analysis is possible only if the timed net is bounded (this property, in general undecidable, carries over from the underlying untimed net)[6] and this is the best result known to date (at least to us.) Luckily, boundedness is in general a well accepted requirement for workflow models. In our model, each transition will be associated with a *firing delay interval.*

Definition 1. A **timed net** is a system, $net = \langle pSet, tSet, fRel, lab, delay\rangle$, which consists of:

- disjoint, finite and non empty sets *pSet* of *places* and *tSet* of *transitions.*

– the *flow relation* $fRel \subseteq (pSet \times tSet) \bigcup (tSet \times pSet)$ which is such that

$$\underline{domain}(fRel) \bigcup \underline{range}(fRel) = tSet \times pSet.$$
$$\forall t \in tSet,\ \exists p \in pSet\ (p,t) \in fRel$$

– the *labelling* function $lab : tSet \longrightarrow \mathcal{AN}$.
– $delay : tSet \longrightarrow \mathbb{T}$, the *delay interval* function.

Moreover, $\underline{Nets}$ denotes the class of all timed nets.

Note. For $x \in pSet \bigcup tSet$, x is called an element of net. The set of elements of net is denoted $\underline{Elem}(net)$. The *output set* of an element x, denoted $\underline{out}_{net}(x)$, is the set $\{y \mid (x,y) \in fRel\}$. The *input set* of x, denoted $\underline{inp}_{net}(x)$, is the set $\{y \mid (y,x) \in fRel\}$. These notions are extended to sets in the usual manner. The subscript net will be dropped whenever it is clear from the context.

A *path* from x to y in net is a non empty sequence $path = x_1 \ldots x_n$ such that:

$$[\forall i = 1,\ldots,n-1,\ x_i \in \underline{inp}(x_{i+1})]\ \&\ [x = x_1\ \&\ y = x_n]$$

$\underline{Path}(x,y,net)$ will be used to denote the class of all paths from x to y in net. Note here that $\underline{Path}(x,x,net)$ is always a non empty set.

For $t \in tSet$, if $delay(t) = [0]$ then t is said to be an *instantaneous transition.* If $label(t) = \lambda$, then t is said to be *silent.* If all the transition are instantaneous, then net is said to be *untimed.*

Definition 2. A **timed flow net** is a system, $flow = \langle net; s_{in}, s_{out}\rangle$, which consists of:

– $net \in \underline{Nets}$, the *underlying timed net* of $flow$, denoted $flow^\circ$, which is such that

$$\forall x \in \underline{Elem}(net),\ \underline{Path}(s_{in}, x, net) \neq \emptyset\ \&\ \underline{Path}(x, s_{out}, net) \neq \emptyset. \tag{1}$$

– $s_{in} \in pSet$ is the *entry place*, and $s_{out} \in pSet$ is the *exit place* which are such that

$$\underline{inp}(s_{in}) = \emptyset\ \&\ \underline{out}(s_{out}) = \emptyset. \tag{2}$$

Moreover, $\underline{FNets}$ denotes the class of all timed flow nets.

Note. Condition (1) states that every element is in a path linking the entry to the exit place. Condition (2) states that the entry place has no incoming edges and that the exit place has no outgoing edges. Note here that the entry and the exit places are necessarily unique.

The *interface* of $flow$, denoted $\underline{interface}(flow)$, is the set $\{s_{in}, s_{out}\}$. Any place $p \notin \underline{interface}(flow)$, is said to be *internal.* $\underline{PInterior}\,(flow)$ denotes the set of all internal places and $\underline{Interior}\,(flow) = \underline{Elem}(flow) - \underline{interface}(flow)$.

Example 1. Consider an office procedure for order processing within a typical electronics company. When a customer requests by mail, or in person, an electronic part, this is the beginning of a job. An order form is filled out by the clerical staff (*order_entry* activity).) The order form is routed in parallel to the finance department for customer credit check (*credit_check* activity,) and to the inventory department for inventory check (*inventory_check* activity.) The finance agent files a customer credit report with the collection agency, records its findings in the order form and sends the order form to the sales department. The inventory agent checks the availability of the goods, and files a report to be sent along with the order form to the sales department. After evaluating the reports (*evaluation* activity,) the order is either rejected and a rejection letter (*notify_reject* activity) is sent to the customer, or the order is submitted to the sales manager for approval (*approval* activity.) Once the order is approved, it is routed in parallel to the shipping and the billing departments. The shipping department will actually cause the parts to be sent to the customer (*shipping* activity;) the billing department will see that the customer is sent a bill, and that it is paid (*billing* activity.) Finally, a log with a description of the order processing is created by the system (*archiving* activity.)

Figure 1 depicts a version of this order processing procedure model by the timed flow net $OrderProc_1$. The *order_entry* activity is modeled by the transition t_{oe}. The *credit_check* activity is modeled by t_{cc}. The *inventory_check* is modeled by t_{ic}. The *evaluation* activity is modeled by t_{ev}. The *approval* activity is modeled by t_{ap}. The *shipping* activity is modeled by t_{sh}. The *billing* activity is modeled by t_{bi}. The *archiving* activity is modeled by t_{ar}. The *notify_reject* activity is modeled by t_{nr}. t_1 and t_3 are used to model the parallel fork construct. t_2 and t_4 are used for the parallel join construct. For the sake of simplicity, the time information associated with an activity reflects the total execution time (i.e. from the time a work unit arrives to the desktop of an actor, to the time its is completed.) This simplistic view is used for illustration purposes only.

1.2 The Semantics of Timed Flow Nets

The question as to how to deal with marking extension has also given rise to at least two proposals. The original one [14, 6] extends the (untimed) marking with a set of dynamic firing intervals. The new one [22, 5, 10, 11] tends to lean toward the Colored Petri net approach[12]; a timed token contains time information (a time-stamp and/or a time interval) which in general is related to the creation/availability of the token.

Definition 3. Let $net \in \underline{Nets}$.

- A **timed token** over *net* is a system, $tk = \langle loc, av\_time \rangle$, which consists of:
  - $loc \in pSet$, the *location* of tk.
  - $av\_time \in \mathbb{Q}^+$, the *availability timestamp* of tk.

Moreover, $\underline{Tks}(net)$ denotes the class of all timed tokens over *net*.

- A **marking** of *net* is a distribution $m \subseteq \underline{Tks}(net)_{MS}$.
- $\underline{Mark}(net)$ denotes the class of all markings of *net*

Note. These notions carry over to timed flow nets through their underlying timed nets. For $flow \in \underline{FNets}$ and $m \in \underline{Mark}(flow)$, if m consists of a single token residing in s_{in}, then m is said to be an *initial* marking of $flow$. Likewise, if m consists of a single token residing in s_{out}, then m is said to be a *terminal* marking of $flow$. $\underline{1}_{flow}$ (resp. $\overline{1}_{flow}$) denotes the class of all initial (resp. terminal) markings of $flow$. The subscript $flow$ will be dropped whenever it is clear from the context. $\underline{1}^*$ denotes the initial marking whose only token has a timestamp 0.

For $P \subseteq pSet$, $m{\downarrow_P}$ denotes the marking m' obtained from m by removing all tokens not located in P. Furthermore, if $m' = m$, then m is said to be a P-marking and if additionally every place in P has a single token under m, then m is said to be a *simple* P-marking. $\underline{PMark}(P)$ denotes the class of all P-markings and $\underline{PSMark}(P)$ denotes the class of all simple P-markings.

A transition fires by *consuming* one token from each of its input places, and produces one token in each of its output places after a delay has elapsed, this delay is relative and prescribed by the firing delay interval of the transition. Tokens are consumed in order of their availability timestamps. In other words, if more than one token resides in the input place of a transition, then a token with the smallest availability timestamp is consumed first.

In our model, this is represented by events. An event description includes a transition to be fired, along with its *enabling time* and its *firing delay* The enabling time of an event is the maximum availability time associated with the consumed tokens. Thus, the event with the smallest enabling time fires first (*FIFO*.) If more than one event is enabled, then the choice is non deterministic. An event fires (i.e. consumes tokens) at time x iff no consumed token has a timestamp greater than x. Thus, the *firing time* of an event is greater or equal to its enabling time. For the sake of simplicity, we assume that the firing time and the enabling time of an event coincide (*eager firing*).

Example 2. Consider the marked timed net depicted in figure 2. Place p_1 has 1 token tk_1 with a time-stamp of 2, place p_2 has two tokens tk_2 with a time-stamp of 1 and tk_3 with a time-stamp of 4, p_3 has 1 token tk_4 with a time-stamp of 3. The transition t_1 is enabled at 2 because the only two tokens it would consume at this point are tk_1 and tk_2 (and not tk_3 because tk_2 has a smaller time-stamp.) and 2 is the maximal time-stamp of the consumed tokens. The transition t_2 is enabled at 3 because the only two tokens it would consume are tk_2 and tk_4. Thus, t_2 fires first, and produces one token tk_5 in p_4 with a time-stamp of 5.25; any time-stamp between $2 + 3 = 5$ and $2 + 4 = 6$ is valid. Under the new reached marking, the transition t_2 is enabled at 4 (note here that it has changed). t_2 consumes the tokens tk_3 and tk_4 and produces a token tk_6 in p_5 with a time-stamp of 5.75; any time-stamp between $4 + 1 = 5$ and $4 + 2 = 6$ is valid. Then t_3 fires at 5.75; tk_5 and tk_6 are consumed and a token is produced in p_6 with a time-stamp of 7.

Despite its simplicity, the model is very expressive and allows for a range of time interpretations including activity durations and timeouts as illustrated next.

Example 3. Consider the case of a workflow procedure specification within a parts factory which reads as follows: *Any part which is not processed by the mold_part activity one hour after it has entered the system, has to be recycled.* There are many other situations which require some timer modeling capabilities. A version of this workflow procedure is depicted in figure 3. Here, we have extended our model with priorities (we will discuss how our model can accommodate such an extension.) Assume that a part enters the system at 9:20. The part is checked by the *part_check* activity in 10 mn. Then, a timer is set to expire at 10:30 through the firing of the transition *set_timer*. The firing of the transition *timer_expired* signals the expiration of the timer. If a token arrives at p_2 before or at 10:20, the transitions *timer_ok* and *timer_expired* are set to fire at 10:20. Since *stop_ok* has a higher priority, it will fire (hence disabling *timer_expired*) and the activity *part_mold* is initiated at exactly 10:30. If the token arrives at p_2 after 10:30, say at 10:35, then *timer_expired* fires at 10:20; the token in p_2 is consumed and at the same time a token is produced in p_3. The part will not be processed by *mold_part*; instead it is recycled through the activity *part_recycle* at 10:35.

Example 4. To model activities with durations, we use the construction shown in figure 3. The immediate transition $\underline{t}$ reflects the initiation of the activity t. A token in p_{exec} means that the activity is in progress. The completion of the activity is modeled by the firing of the transition $\overline{t}$.

Definition 4. Let $net \in \underline{Nets}$.

- An **event** over net is a system, $evt = \langle tr, en\_time, fr\_delay \rangle$, which consists of:
  - $tr \in tSet$, the *underlying* transition of evt.

- $en\_time \in \mathbb{Q}^+$, the *enabling time* of *evt*.
- $fr\_time \in \mathbb{Q}^+$, the *firing delay* of *evt*, which is such that $fr\_delay \in delay(tr)$.

- The *completion time* of *evt*, denoted $\underline{cpl\_time}(evt)$, is $en\_time + fr\_delay$.
- $\underline{Evts}\,(net)$ denotes the class of all events over *net*.

Note. Let $w \in (\underline{Evts}\,(net))^*$, w is a *valid* sequence iff

$$\forall 1 \leq i < \underline{lgth}(w), w_i.en\_time \leq w_{i+1}.en\_time$$

For $w \in \underline{VSeq}\,(net)$, the *start time* of w, denoted $\underline{start}\,(w)$, is $w[1].en\_time$. The *end time* of w, denoted $\underline{end}\,(w)$, is the the maximal completion time of its events. The *time length* of w, denoted $\underline{time\_lgth}(w)$, is $\underline{end}\,(w) - \underline{start}\,(w)$. The *lifetime* of w, denoted $\underline{life}\,(w)$, is the time interval For $T \subseteq tSet$, $\underline{project}(w, T)$, denotes the valid sequence of *net* obtained by removing all the events *evt* such that $evt.tr \notin T$. These notions are lifted to the level of activity names. Thus, for $w \in \underline{VSeq}\,(net)$, $\underline{label}(w)$ denotes the sequence obtained from w by applying the labeling functions to the underlying transitions after erasing all events whose underlying transitions are silent.

Sometime, we will be interested in the sequence of transition firings. In our terminology this is referred to as a *trace*. Thus, for $w \in \underline{VSeq}\,(net)$, the trace of w, denoted $\underline{trace}(w)$, is the sequence obtained from w by forgetting about the enabling times and firing delays of w. A sequence $w' \in tSet^*$ is a (m, m')-trace iff there exists an (m, m')-firing sequence w such that $\underline{trace}(w) = w'$. $\underline{Trace}(net, m, m')$ will be used to denote the language of $(m'm')$-traces of *flow*. $\underline{label}(w')$ will be used to denote the sequence obtained from w' by considering the labeling of $w'[i]$'s.

Definition 5. Let $net \in Nets$, let $m \in \underline{Mark}(net)$ and let $evt \in \underline{Evts}\,(net)$.

- *evt* is **enabled** under m in *net*, written $m\,[[evt\rangle_{net}$, iff

$$\exists tk_{in} \in \underline{PSMark}(\underline{inp}(evt.tr)), tk_{in} \subseteq m. \tag{3}$$

- *evt* is **time enabled** under m in *net*, written $m\,[evt\rangle_{net}$, iff the following conditions hold:

$$\begin{array}{l} m\,[[e\rangle \\ evt.en\_time = max\,\{tk.av\_time \mid tk \in tk_{in}\} \\ \forall tk' \in m,\ \forall tk \in tk_{in}, \\ \quad loc(tk') = loc(tk) \Rightarrow av\_time(tk) \leq av\_time(tk') \\ \forall evt' \in \underline{Evts}\,(net)\,, \\ \quad m\,[[evt'\rangle_{net} \Rightarrow evt.en\_time \leq evt'.en\_time \end{array} \tag{4}$$

In this case, the **firing** of *evt* under m in *net leads* to the marking m', written $m\,[evt\rangle_{net}\, m'$, where

$$m' = (m - tk_{in}) \bigcup \{\langle p, \underline{cpl\_time}(evt)\rangle\, | p \in \underline{out}(evt.tr)\} \tag{5}$$

- $\xrightarrow{net} \subseteq \underline{Mark}(net) \times \underline{Evts}\,(net)^* \times \underline{Mark}(net)$ denotes the event firing relation associated with *net* and is given by:

$(m, w, m') \in \xrightarrow{net}$ iff $m = m'$ & $w = \lambda$ or the following condition is true:
$\quad \exists m'' \in \underline{Mark}(net), \exists evt \in \underline{Evts}\,(net)\,,$
$\quad\quad \left[w = w' \bullet evt \;\&\; (m, w', m'') \in \xrightarrow{net} \;\&\; m''\,[evt\rangle_{net}\, m'\right]$

Condition(3) state that an event is enabled if there are enough tokens to be consumed by the underlying transition. Conditions(4) state that an event fires as soon as it is enabled, that the tokens are consumed in order of their availability timestamps, that events with the smallest enabling time fire first. Condition (5) states that tokens are produced after the firing delay has elapsed.

Extensions. (Priorities, Generalized Time Model) Our timed flow net model can be extended in many ways; including the use of

1. **priorities** to resolve time conflicts between enabled events. That is to say that when two simultaneously enabled events are "racing" to consume the same token, the one with higher priority is time enabled. This extra time enabling condition is expressed as follows:

$$\forall evt' \in \underline{Evts}(net), \begin{bmatrix} Enableddmevt'net \\ evt.en\_time = evt'.en\_time \\ \underline{inp}(evt.tr) \bigcap \underline{inp}(evt'.tr) \neq \emptyset \end{bmatrix} \Rightarrow evt.tr.priority \leq evt'.tr.priority \qquad (6)$$

2. a **random** time **distribution** or a **generalized** time **expression** to compute an event's firing delay. Because this computation is hidden from our firing semantics, such extensions can be accommodated without any difficulty.

As this work progresses, we will examine the ramifications of such model extensions on our results. Occasionally, we will use priorities in our illustrated examples

Note. We will write $m\,[w\rangle_{net}\, m'$ instead of $(m, w, m') \in \xrightarrow{net}$. net will be dropped from all notations whenever it is clear from the context. The sequence w is called a *(m, m')-firing sequence.* $\underline{Fire}(net, m, m')$ denotes the class of all (m, m')-firing sequences of *net*.

These notions carry over to timed flow nets through their underlying timed nets. For a timed flow net, $flow$, an *execution* is a firing sequence which starts at an initial marking. A *schedule* is an execution which ends at a terminal marking. $\underline{Exec}(flow)$ (resp. $\underline{Sched}(flow)$) denotes the language of all executions (resp. schedules.) of $flow$.

The execution proceeds by processing what is commonly known as a *job*. Each job has a name which uniquely identifies the job at any given time, a flow which identifies the workflow procedure which is operating upon the job. It also has a history of the event firings which have so-far taken place as part of the execution of the job. In the sequel, we assume that jobs do not interfere with each other. This assumption is carried out using the so-called *copy rule.*

Definition 6. Let $flow \in \underline{FNets}$. A **job** over $flow$ is a system, $job = \langle name, state_{in}, state_{cr}, hist\rangle$, which consists of:

- $name \in \mathcal{JN}$, the *name* of the job.
- $state_{in} \in \underline{1}_{flow}$, the *initial state* of the job.
- $state_{cr} \in \underline{Mark}(flow)$, the *current state* of the job.
- $hist \in \underline{Fire}(flow, state_{in}, state_{now})$, the *history* of the job.

$\underline{Jobs}(flow)$ denotes the class of all jobs over $flow$.

Example 5. Consider the timed flow net $OrderProc_1$ introduced in example 1 (see figure 1) and the job *Jones* which enters the system at 8:00 a.m. the initial state is $m_0 = \{tk_{in} = (p_{in}, 8{:}00)\}$.

At 8:00, the job is submitted to the *order_entry* activity t_{oe} for processing, the processing takes about 3mn. This is modeled by the firing of the the event $oe = \langle t_{oe}, 8{:}00, 00{:}03\rangle$ under m_0. This firing leads to the marking m_1 where $m_1 = \{tk_1 = \langle p_1, 8{:}03\rangle\}$.

At 8:03, the job is routed in parallel to the *credit_check* activity t_{cc} and to the *inventory_check* activity t_{ic}. In our model, this is reflected by the firing of the event $rte = \langle t_1, 8{:}03, 00{:}00\rangle$ under m_1. Here, we assume that the routing is instantaneous (i.e. the routing delay is 0.) The firing leads to the marking $m_2 = \{tk_2 = \langle p_2, 8{:}03\rangle, tk_3 = \langle p_3, 8{:}03\rangle\}$.

At 8:03, the activities t_{cc} and t_{ic} are initiated. t_{cc} is completed $8mn$ after it is initiated, whereas t_{ic} is done in $5mn$. In our model, this corresponds to the firing under m_2 of the events $cc = \langle t_{cc}, 8{:}03, 00{:}8\rangle$ and $ic = \langle t_{ic}, 8{:}03, 00{:}05\rangle$. These events firings can be recorded in any order. Thus, we may have either $w_1 = ic.cc$ or $w_2 = cc.ic$. If one is interested in resolving ordering ambiguity using the completion times of event firings, then w_1 is more appropriate. Assuming that this is the case, the marking reached after completion of ic is $m_3 = \{tk_2, tk_5 = \langle p_5, 8{:}08\rangle\}$, and the marking reached after the completion of cc is $m_4 = \{tk_4 = \langle p_4, 8{:}11\rangle, tk_3\}$.

At 8:11 (i.e. after completion of cc and ic), tk_4 and tk_5 are consolidated, then the job is submitted to the *evaluation* activity t_{ev} for processing. This is modeled by the firing of the event $mrg = \langle t_2, 8{:}11, 0\rangle$. Here, we assume that the join or consolidation action is instantaneous. This firing leads to the marking $m_5 = \{tk_6 = \langle p_6, 8{:}11\rangle\}$.

At 8:11, the evaluation process starts and takes about 4mn. This is modeled by the firing of the event $ev = \langle t_{ev}, 8{:}11, 00{:}04\rangle$ under m_5. This firing leads to the marking $m_6 = \{tk_7 = \langle p_7, 8{:}15\rangle\}$. The order is rejected because not all parts ordered by the customer are in stock.

At 8:15, the system generates a customized rejection letter and a notification letter is sent to the customer. This is modeled by the firing of the event $nr = \langle t_{nr}, 8{:}15, 00{:}03\rangle$ under m_6 which leads to the marking $m_7 = \{tk_{13} = \langle p_{13}, 8{:}18\rangle\}$ (assuming that the process takes 3mn.)

At 8:18, the *archiving* activity t_{ar} is initiated; the order log is created and stored in the archives. This is modeled by the firing of the event $ar = \langle t_{ar}, 8{:}18, 00{:}02\rangle$ under m_7 which leads to the terminal marking $m_{out} = \{tk_{out} = \langle p_{out}, 8{:}20\rangle\}$ (assuming that it takes about 2mn.)

At 8:20, the case *Jones* is completed and exits the system. The sequence $sched = oe.rte.ic.cc.mrg.ev.nr.ar$ is a schedule, its life time $\underline{life}(sched) = [8{:}00, 8{:}20]$, and its trace is

$$\underline{trace}(sched) = order\_entry.inventory\_check.credit\_check.eval.notify\_reject.archive.$$

1.3 Sound flow nets

In [20], Van der Aalst introduces workflow nets and the notion of sound workflow nets. The author shows that the soundness property is decidable by linking it to the boundedness and liveness of the net obtained by adding a transition which links the entry to the exit place.

Definition 7. Let $flow$ be a timed flow net.

$flow$ is **sound** iff the following conditions hold:

$$\begin{array}{l} \forall m \in \underline{Reach}(flow, \underline{1}^*),\ \underline{Reach}(flow, m) \bigcap \overline{1} \neq \emptyset. \\ \forall t \in tSet, \exists m \in \underline{Reach}(flow, \underline{1}^*), \exists evt \in \underline{Evts}(flow) \\ \quad m\,[evt\rangle\ \&\ evt.tr = t \end{array} \qquad (7)$$

$\underline{SFNets}$ denotes the class of all sound timed flow nets.

The first condition in (7), referred to as the *proper termination* condition, ensures that each execution of *flow* will end in a a terminal state. The second condition ensures that *flow* has no *dead transitions.*

Note. For $flow \in \underline{FNets}$, $flow^*$ is denotes the marked timed net obtained from *flow* by adding a silent, immediate transition which connects the exit place of *flow* to it input place, and whose marking coincide with $\underline{1}^*_{flow}$.

Unfortunately, soundness is undecidable. Indeed, proper termination and absence of dead transitions are linked to reachability analysis, and as we have previously mentioned timed flow nets resist in general any kind of reachability analysis.

Furthermore, the linkage to boundedness and liveness properties, as established by Van der Aalst, is broken. Formally,

Proposition 1. *Let* $flow \in \underline{FNets}$. *The following property holds:*

$$flow \in \underline{SFNets} \Rightarrow flow^* \text{ is live and bounded.} \tag{8}$$

The converse of (8) does not in general hold. To see that, consider the timed flow net, $flow_1$, depicted in figure 4. Clearly, $flow_1$ is not sound; there is a schedule whose underlying trace is $t_1t_2t_3t_4t_5t_6$ which leads to the marking under which both the exit place s_{out} and the place p are both marked. On the other hand, $flow_1^*$ is live and 1-safe. To see the safeness, note that the only place which may not be 1-safe is p (consider the untimed structure). However, note that the first iteration of $flow_1^*$ will result in both p_{in} and p marked, and that at the end of the nth iteration, one of the following things may occur:

1. if p is not initially marked, then it will be marked with 1 token.
2. if p is initially marked, then the token is either flushed (t_8t_9) or kept (t_8t_{10} or $t_1t_2t_3t_4t_7t_5t_6$).

The soundness property does not carry over to timed flow nets from their underlying (untimed) flow nets. To see that consider the timed flow net $flow_2$ depicted in figure 4. Clearly, the underlying (untimed) flow net is sound, but the timed version is not. After firing t_1, (p_1, p_2) becomes a sort of home marking and (p_4, p_5) is not reachable. However, the soundness property is decidable on timed flow nets whose underlying untimed flow nets are bounded. This requirement is generally acceptable.

2 Process Structural Changes: The Static Model

We adapt the model of change from [9] to accommodate the temporal nature of timed flow nets and to analyze the change correctness on a job basis. Like in the previous work, we shall focus on a special type of workflow procedure change; namely the *structural* change. Structural means that the change is made to the structure of the procedure (as opposed to the data-value). A change is either *dynamic* or *static* with respect to a job; dynamic means that the change is applied while the job is in progress, otherwise if the change is applied before the job starts executing or after its completion, then the change is static. Another classification could be made based on the scope of the change; if the change is applicable to a specific set of jobs (i.e. execution instances,) then the change is referred to as an *instance change*, otherwise it is said to be a *class change*. Examples of instance changes include exceptions. Re-engineering plans are in general considered as instance changes before the cut-off or roll-out date is reached and class changes onward. Critical changes such as fixing hard bugs or related to mission critical systems are considered as class changes.

2.1 Timed Flow Nets Replacements

In a nutshell, our model of structural change is driven by a well-defined discipline which makes its analysis more manageable. This discipline is articulated around the selection of the *change regions* and is based upon the principle of *change locality*.

The *old change region*, denoted *oldRegion*, contains all the activities of the old timed flow net, referred to herein as the *old net* and denoted *oldNet*, which are involved in the change (e.g. deleted, reorganized etc...). This means that when selecting the old region, places connected to these activities as well as the connecting edges are made part of the old region. The *new change region*, denoted *newRegion*, embodies the alterations that the old region undergoes as a result of the change. In order to make the analysis of the change more manageable, The scope of the change should be as much as possible limited to the change regions; this requirement is referred to as the the principle of change locality. In other words, the selection of the old change region minimizes its interaction with its context. This interaction is structurally maintained solely by the interface of the old region, and is reduced to tokens exchange; the context supplies tokens to the old region for consumption (through its input place) and consumes the tokens produced by the old change region in its output place. The old change region is said to be a *closed subnet* of the old net, written $\underline{closed}(oldRegion, oldNet)$.

Definition 8. A **replacement pair** is a system $\delta = \langle oldRegion, newRegion \rangle$ which consists of two timed flow nets *oldRegion*, the *old change region* of δ, and *newRegion*, the *new change region* of δ, which are such that

$$\begin{array}{l} oldRegion.s_{in} = newRegion.s_{in} \\ oldRegion.s_{out} = newRegion.s_{out} \\ \underline{Elem}(oldRegion) \bigcap \underline{Elem}(newRegion) = \underline{interface}(oldRegion) \end{array} \qquad (9)$$

$\underline{ReplPairs}$ denotes the class of all replacement pairs.

Condition (9) states that the old and the new region have the same entry and the same exit places. Furthermore, these are the only shared elements.

Definition 9. Let $net_1, net_2 \in \underline{Nets}$.

– net_1 and net_2 are **disjoint**, written $\underline{disjoint}(net_1, net_2)$, iff

$$\underline{Elem}(net_1) \bigcap \underline{Elem}(net_2) = \emptyset.$$

– net_1 is a **subnet** of net_2, written $\underline{subnet}(net_1, net_2)$, iff

$$\begin{gathered} pSet_1 \subseteq pSet_2 \;\&\; tSet_1 \subseteq tSet_2 \\ fRel_1 = fRel_2 \bigcap [(pSet_1 \times tSet_1) \bigcup (tSet_1 \times pSet_1)] \\ lab_1 = lab_2 {\downarrow}_{tSet_1} \\ delay_1 = delay_2 {\downarrow}_{tSet_1} \end{gathered}$$

Definition 10. Let $flow \in \underline{FNets}$ and let $net \in \underline{Nets}$ such that

$$\underline{subnet}(flow, net)$$

$flow$ is a **closed** in net, written $\underline{closed}(flow, net)$, iff

$$\forall x \in \underline{Interior}\,(flow)\ \left[\underline{inp}_{net}(x) \bigcup \underline{out}_{net}(x) \subseteq \underline{Elem}(flow)\right]$$

In this case, the *context* of $flow$ in net, denoted $\underline{ctxt}(flow, net)$, is the timed net net' defined as follows:

$$\begin{aligned} &pSet' = (pSet_{net} - pSet_{flow}) \bigcup \underline{interface}(flow) \\ &tSet' = tSet_{net} - tSet_{flow} \\ &fRel' = fRel \bigcap (pSet' \times tSet' \bigcup tSet' \times pSet') \\ &lab' = lab_{net} {\downarrow}_{tSet'} \\ &delay' = delay_{net} {\downarrow}_{tSet'} \end{aligned}$$

After the change regions are selected properly, the replacement may take place, resulting in a new flow, referred to as the *new net* and denoted $newNet$. The new net is obtained from the old net by:

1. plugging the new change region into the old net by using the shared interface as sockets.
2. removing all the internal elements of the old change region from the resulting flow net.

In order to formalize this rewriting mechanism, we will define a few net operations. The disjoint sum of two nets is the net obtained by putting them side by side. The fusion of two places removes one place and has the other place inherit the connections of the removed place. Formally,

Definition 11. Let net_1, net_2 be disjoint nets.

The **disjoint sum** of net_1 and net_2, denoted $net_1 \oplus net_2$, is the net, net', defined as follows:

$$\begin{gathered} pSet = pSet_1 \bigcup pSet_2 \\ tSet = tSet_1 \bigcup tSet_2 \\ fRel = fRel_1 \bigcup fRel_2 \\ label = label_1 \bigcup label_2 \\ delay = delay_1 \bigcup delay_2 \end{gathered}$$

Definition 12. Let $net \in \underline{Nets}$ and let $p \neq q \in pSet$.

The (p, q)-$\underline{\text{fusion}}$ of net, denoted $\theta_{p,q}(net)$, is the timed net net' obtained as follows:

$$\begin{gathered} pSet' = pSet - \{q\} \\ tSet' = tSet \\ fRel' = (fRel \bigcap (pSet' \times tSet' \bigcup tSet' \times pSet')) \\ \bigcup (\{p\} \times \underline{out}(q)) \bigcup (\underline{inp}(q) \times \{p\}) \\ label' = label \\ delay' = delay \end{gathered}$$

Note. This notion is expanded to the case where $p = q$ through the equality $\theta_{p,p}(net) = net$. It is also lifted to an arbitrary finite sequence σ of pairs as follows:

$$\begin{aligned} &\theta_\lambda(net) = net \\ &\theta_{w.w'}(net) = \theta_w(\theta_{w'}(net)) \end{aligned}$$

Definition 13. Let $net \in Nets$. Let $\delta \in \underline{ReplPairs}$.

- δ is **applicable** to net, written $net \rightsquigarrow_\delta$, iff the following conditions holds:

$$\begin{array}{l}\underline{closed}(oldRegion, net)\\ \underline{Elem}(newRegion) \bigcap \underline{Elem}(\underline{ctxt}(oldRegion, oldNet)) = \underline{interface}(newRegion)\end{array}$$

- In this case, the **application** of δ to net leads to the timed net net' defined as follows:

$$net' = \theta_{\sigma'}(\alpha_\sigma(newRegion) \oplus \underline{ctxt}(oldRegion, net)) \tag{10}$$

where

$$\begin{array}{l}\sigma = \{(newRegion.s_{in}, p), (newRegion.s_{out}, q)\}\\ \sigma' = (newRegion.s_{in}, p) \bullet (newRegion.s_{out}, q)\\ p, q \notin \underline{Elem}(newRegion) \bigcup \underline{Elem}(net)\end{array}$$

This will be denoted $net \rightsquigarrow_\delta net'$. The system $rep = \langle net, \delta, net' \rangle$ is a *replacement step*

Note. We will write $oldNet \rightsquigarrow_\delta$ to say that δ is applicable to $oldNet$, and $oldNet \rightsquigarrow_\delta newNet$ to say that the timed flow net, $newNet$, is obtained from the timed flow net, $oldNet$, by applying the replacement mechanism as previously outlined. The tuple $repl = (oldNet, \delta, newNet)$ will be referred to as a *replacement step*. $\underline{ReplPairs}$ will denote the class of all replacement pairs. The formal definitions are given in the appendix.

Example 6. In the case of our order processing procedure, it has been decided to initiate *billing* and *shipping* in sequence, instead of concurrently as it was previously done. Moreover, this change should not affect the completion times previously reached. The general consensus was to speed up these two activities by acquiring high end systems. The old and new change regions as well as the old and the new nets are depicted in figure 5.

The introduction of the replacement mechanism leads to the natural question as to how it affects the semantics and the properties of a flow net. For the case of untimed flow nets, this issue has been settled in [9, 20]. In what follows, we extend these results for the soundness and other properties including *time*, *schedule* and *trace approximations*, which are introduced next.

Definition 14. Let $flow_1, flow_2 \in \underline{FNets}$.

- $flow_1$ is a **time approximation** of $flow_2$, written $flow_1 \sqsubseteq_{Time} flow_2$, iff the following condition holds:

$$\forall w_1 \in \underline{Sched}(flow_1),\ \exists w_2 \in \underline{Sched}(flow_2),\ \underline{life}(flow_1) = \underline{life}(flow_2)\,.$$

- $flow_1$ is a **schedule approximation** of $flow_2$, written $flow_1 \sqsubseteq_{Sched} flow_2$, iff the following condition holds:

$$\begin{array}{l}\forall w_1 \in \underline{Sched}(flow_1),\ \exists w_2 \in \underline{Sched}(flow_2),\\ \quad \underline{life}(flow_1) = \underline{life}(flow_2)\ \&\ \underline{label}(w_1) = \underline{label}(w_2).\end{array}$$

- $flow_1$ is a **trace approximation** of $flow_2$, written $flow_1 \sqsubseteq_{Trace} flow_2$, iff the following condition holds:

$$\begin{array}{l}\forall w_1 \in \underline{Sched}(flow_1),\ \exists w_2 \in \underline{Sched}(flow_2),\\ \quad \underline{life}(flow_1) = \underline{life}(flow_2)\ \&\ \underline{trace}(\underline{label}(w_1)) = \underline{trace}(\underline{label}(w_2)).\end{array}$$

In order to carry out our work, we need to define some auxiliary notions, namely the *augmentation*, *separation*.

Note. In the remaining of this section,we make the following assumption:

$$\forall flow \in \underline{FNets},\ p_{flow}, \overline{p_{flow}}, t_{flow}, \overline{t_{flow}} \notin \underline{Elem}(flow). \tag{11}$$

Let $flow \in \underline{FNets}$, $\underline{aug}(flow)$ is the flow net obtained from $flow$ by:

1. renaming s_{in} into $\underline{p_{flow}}$ and s_{out} into $\overline{p_{flow}}$.
2. adding an immediate silent transition $\underline{t_{flow}}$ with s_{in} as the only input place and $\underline{p_{flow}}$ as the only output place.
3. adding an immediate silent transition $\overline{t_{flow}}$ with $\overline{p_{flow}}$ as the only input place and s_{out} as the only output place.

Note here that the construction preserves the interface. If $\underline{closed}(flow, flow')$ for some $flow' \in \underline{FNets}$, then $\underline{sep}(flow, flow')$ is the flow net obtained from $flow'$ by replacing $flow$ with $\underline{aug}(flow)$.
The notion of augmentation will be extended to our replacement mechanism as follows:
if $\delta = \langle oldRegion, newRegion\rangle$ is a replacement pair, then $\underline{aug}(\delta) = \langle \underline{aug}(oldRegion), \underline{aug}(newRegion)\rangle$.
If $repl = \langle oldNet, \delta, newNet\rangle$, then $\underline{aug}(repl) = \langle \underline{sep}(oldRegion, oldNet), \underline{aug}(\delta), \underline{sep}(newRegion, newNet)\rangle$.

2.2 Seriality and Segment Iteration

Definition 15. Let $flow_1, flow_2 \in \underline{FNets}$ such that:

$$\underline{closed}(flow_1, flow_2) \ \& \ flow_1 \in \underline{SFNets}.$$

$flow_1$ is **serial** in $flow_2$, written $\underline{serial}(flow_1, flow2)$, iff the following condition holds:

$$\forall m \in \underline{Reach}(flow_2', \underline{1}^*),\ \forall w \in \underline{Trace}(flow_2', \underline{1}^*, m'),\ \underline{project}(w, \Delta) \in \underline{Suffix}((\underline{t_{flow_1}}.\overline{t_{flow_1}})^*) \qquad (12)$$

where $flow_2' = \underline{sep}(flow_1, flow_2)$ & $\Delta = \left\{\underline{t_{flow_1}}, \overline{t_{flow_1}}\right\}$.

Condition 12 precludes any token from entering $flow_1$ while it is in progress. In the definition above, we have resorted to the use of $\underline{sep}(flow_1, flow_2)$ for two reasons. First, for ease of formulation; note here that $\underline{t_{flow_1}}$ (resp. $\overline{t_{flow_1}}$) reflects the situation where a token enters (resp. exits from) $flow_1$. Second, to deal with the peculiar case where $flow_1$ has a schedule which consists of a single event.

Example 7. Consider the sound flow net $flow_i$ for $i = 1 \dots 4$ depicted in figure 6. $flow_3 = \underline{aug}(flow_1)$, $flow_4 = \underline{sep}(flow_1, flow_2)$ and $\underline{cpl\_time}(flow_3) = [0, 2]$. Let $w_{init} = \langle t_1, 0, 0\rangle \langle t_2, 0, 4\rangle \langle t_3, 0, 1\rangle$ and let m_4 the marking of $flow_4$ such that $\underline{1}^* [w_{init}\rangle\, m_4$. Clearly, m_4 has two tokens which reside in p_3; namely $tk_1 = \langle p_3, 1\rangle$ and $tk_2 = \langle p_3, 4\rangle$. The time separating the arrivals of these tokens to p_3 is 3 and thus, greater than the maximal completion time of $flow_3$. Moreover, this property holds for all markings $m_4 \in \underline{Reach}(flow_4, \underline{1}^*)$. This means that $\underline{serial}(flow_1, flow_2)$ holds.

Next, consider the sound flow nets $flow_5$, $flow_6$ and $flow_7$ depicted in figure 7. $flow_6 = \underline{aug}(flow_5)$, $flow_7 = \underline{sep}(flow_5, flow_2)$ and $\underline{cpl\_time}(flow_5) = [4, 7]$. Let

$$w_{cont} = \left\langle \underline{t_{flow_5}}, 1, 0\right\rangle \langle t_4, 1, 3\rangle \langle t_{10}, 1, 1\rangle \langle t_{11}, 2, 3\rangle \left\langle \underline{t_{flow_5}}, 4, 0\right\rangle \text{ and } w = w_{init}.w_{cont}.$$

Clearly, $w \in \underline{Fire}(flow_7, m_4, m_7)$ where $m_7 = \left\{\langle p_4, 4\rangle, \left\langle \underline{p_{flow_5}}, 4\right\rangle, \langle p_{12}, 5\rangle, \langle p_7, 4\rangle\right\}$. Moreover,

$$\underline{project}(w, \Delta) = \underline{t_{flow_5}}.\underline{t_{flow_5}} \notin \underline{Suffix}((\underline{t_{flow_5}}.\overline{t_{flow_5}})^*), \text{ where } \Delta = \left\{\underline{t_{flow_5}}, \overline{t_{flow_5}}\right\}.$$

Thus, w violates the seriality condition. Intuitively, this means that 2 executions of $flow_5$ are in progress at the same time.

Our first results examine the relationship between a flow net and its separation. In a nutshell, the first result states that *the separation of a flow net, $flow_2$, through a sound closed subnet, $flow_1$, "preserves" the semantics of $flow_2$* . This means that *notions such as boundedness, liveness, soundness, and the diverse approximation notions can be investigated in either flow nets.* Formally,

Proposition 2. *Let $flow_1, flow_2 \in \underline{FNets}$ such that:*

$$\underline{closed}(flow_1, flow_2) \ \&\ flow_1 \in \underline{SFNets},$$

let $flow'_1 = \underline{aug}(flow_1)$ and let $flow'_2 = \underline{sep}(flow_1, flow_2)$.

The following properties hold:

$$\forall m'_2 \in \underline{Reach}(flow'_2, \underline{1}^*),\ \exists m_2 \in \underline{Reach}(flow_2, \underline{1}^*),\ m_2 = h(m'_2) \tag{13}$$

$$\forall m_2 \in \underline{Reach}(flow_2, \underline{1}^*),\ \exists m'_2 \in \underline{Reach}(flow_2, \underline{1}^*),\ m_2 = h(m'_2) \ \&\ m_2{\downarrow}_{\{p_{in}\}} = m'_2{\downarrow}_{\{p_{in}\}} \tag{14}$$

where $h : \underline{Tks}\,(flow'_2) \longrightarrow \underline{Tks}\,(flow_2)$ is defined as follows:

$$h(tk) = \begin{cases} tk & \text{if } tk.loc \notin \left\{\underline{p_{flow_1}}, \overline{p_{flow_1}}\right\} \\ \langle p_{in}, \tau\rangle & \text{if } tk = \left\langle \underline{p_{flow_1}}, \tau\right\rangle \\ \langle p_{out}, \tau\rangle & \text{if } tk = \langle \overline{p_{flow_1}}, \tau\rangle \end{cases}$$

$$p_{in} = flow_1.s_{in} \ \&\ p_{out} = flow_1.s_{out}$$

Proof. Let $T_{ctxt} = \underline{ctxt}(flow_1, flow_2).tSet$ and let $T_1 = flow_1.tSet$.

Proof of 13.

By induction on w'_2, we prove that the following property holds:

$$\begin{array}{c}\forall m'_2 \in \underline{Reach}(flow'_2, \underline{1}^*),\ \forall w'_2 \in \underline{Fire}(flow'_2, \underline{1}^*, m'_2) \\ \exists m_2 \in \underline{Reach}(flow_2, \underline{1}^*),\ m_2 = h(m'_2). \qquad (Q_1)\end{array}$$

• *base case:* $w'_2 = \lambda$.
Take $m_2 = m'_2 = \underline{1}^*$. Clearly, Q_1 is satisfied by m'_2, w'_2 and m'_2. End of the case.

• *induction hypothesis:* Assume that Q_1 is satisfied by m'_2, w'_2 and m_2.

• *induction:* Let $evt'_2 \in \underline{Evts}\,(flow'_2)$ such that $m'_2\,[evt'_2\rangle\,\overline{m'}_2$. We will consider the following two cases:

1. $t'_2 = \underline{t_{flow_1}}$ or $t'_2 = \overline{t_{flow_1}}$. From the induction hypothesis, we have $m_2 = h(\overline{m'}_2)$. Thus, Q_1 is also met by $\overline{m'}_2$, $w'_2.evt'_2$ and m_2.
2. $t'_2 \neq \underline{t_{flow_1}}$ and $t'_2 \neq \overline{t_{flow_1}}$. Then from the induction hypothesis, we have $m_2\,[evt'_2\rangle\,\overline{m}_2 \ \&\ \overline{m}_2 = h(\overline{m'}_2)$ for some marking $\overline{m}_2$ of $flow_2$. Thus, Q_1 is met by $\overline{m'}_2$, $w'_2.evt'_2$ and $\overline{m}_2$.

Proof of 14.

By induction on w_2, we prove that the following property holds:

$$\begin{array}{c}\forall m_2 \in \underline{Reach}(flow_2, \underline{1}^*),\ \forall w_2 \in \underline{Fire}(flow'_2, \underline{1}^*, m_2) \\ \exists m'_2 \in \underline{Reach}(flow'_2, \underline{1}^*),\ m_2 = h(m'_2) \ \&\ m_2{\downarrow}_{\{p_{in}\}} = m'_2{\downarrow}_{\{p_{in}\}}. \qquad (Q_2)\end{array}$$

• *base case:* $w_2 = \lambda$.
Take $m_2 = m'_2 = \underline{1}^*$. Clearly, Q_2 is satisfied by m'_2, w'_2 and m'_2. End of the case.

• *induction hypothesis:* Assume that Q_2 is satisfied by m_2, w_2 and m'_2.

• *induction:* Let $evt_2 \in \underline{Evts}\,(flow_2)$ such that $m_2\,[evt_2\rangle\,\overline{m}_2 \ \&\ evt_2 = \langle t_2, en\_time_2, fr\_delay_2\rangle$. We will consider the following cases:

1. $p_{in} \in \underline{inp}_{flow_1}(t_2)$. In other words, $flow_1$ is consuming a token. Let $evt' = \left\langle \underline{t_{flow_1}}, en\_time_2, 0 \right\rangle$. Clearly, $m'_2 \, [evt'.evt_2\rangle \, \overline{m'}_2$ for some $\overline{m'}_2$, and from the induction hypothesis,

$$\overline{m}_2 = h(\overline{m'}_2) \;\&\; \overline{m}_2{\downarrow}_{\{p_{in}\}} = \overline{m'}_2{\downarrow}_{\{p_{in}\}}.$$

Thus, Q_2 is satisfied by $\overline{m}_2$, $w_2.evt_2$ and $\overline{m'}_2$.

2. $p_{out} \in \underline{out}(t_2)$. In other words, the context is consuming a token, say tk, from p_{out}.

   (a) if $tk \in m'_2$, then $m'_2 \, [evt_2\rangle \, \overline{m'}_2$ for some $\overline{m'}_2$, and from the induction hypothesis,

   $$\overline{m}_2 = h(\overline{m'}_2) \;\&\; \overline{m}_2{\downarrow}_{\{p_{in}\}} = \overline{m'}_2{\downarrow}_{\{p_{in}\}}.$$

   Thus, Q_2 is satisfied by $\overline{m}_2$, $w_2.evt_2$ and $\overline{m'}_2$.

   (b) if $tk \notin m'_2$, then there exists a token $tk' \in m'_2$ such that $h(tk') = tk$ & $tk'.loc = \overline{p_{flow_1}}$. In other words, we have to pull the token from $\overline{p_{flow_1}}$. Let $evt' = \langle \overline{t_{flow_1}}, en\_time_2, 0 \rangle$. Clearly, $m'_2 \, [evt'.evt_2\rangle \, \overline{m'}_2$ for some $\overline{m'}_2$, and from the induction hypothesis,

   $$\overline{m}_2 = h(\overline{m'}_2) \;\&\; \overline{m}_2{\downarrow}_{\{p_{in}\}} = \overline{m'}_2{\downarrow}_{\{p_{in}\}}.$$

   Thus, Q_2 is satisfied by $\overline{m}_2$, $w_2.evt_2$ and $\overline{m'}_2$. QED

Proposition 3. *Let $flow_1, flow_2 \in \underline{FNets}$ such that:*

$$\underline{closed}(flow_1, flow_2) \;\&\; flow_1 \in \underline{SFNets},$$

let $flow'_1 = \underline{aug}(flow_1)$ and let $flow'_2 = \underline{sep}(flow_1, flow_2)$.

The following properties hold:

1. $flow_2 \in \underline{SFNets} \Leftrightarrow flow'_2 \in \underline{SFNets}$.
2. $flow_2 \simeq_{Time} flow'_2$.
3. $flow_2 \simeq_{Sched} flow'_2$.
4. $flow_2 \simeq_{Trace} flow'_2$.

Proof. Direct consequence of proposition 2

The next result establishes *a relationship between seriality and 1-safeness* for untimed flow nets.

Proposition 4. *Let $flow_1, flow_2 \in \underline{FNets}$ such that:*

$$\begin{array}{l} \underline{closed}(flow_1, flow_2) \\ flow_1 \in \underline{SFNets}. \\ \forall t \in flow_2.tSet,\ delay(t) = [0]. \end{array}$$

The following property holds:

$$\underline{serial}(flow_1, flow_2) \Leftrightarrow p_{in} \text{ is 1-safe in } (flow_2; \underline{1}^*)$$

where $p_{in} = flow_1.s_{in}$.

Proof. Let $flow'_1 = \underline{aug}(flow_1)$ and let $flow'_2 = \underline{sep}(flow_1, flow_2)$.

• *if-part:* Assume that p_{in} is not 1-safe in $(flow_2; \underline{1}^*)$. By using the equality (14) of proposition 2, we can see that p_{in} is not 1-safe in $(flow'_2; \underline{1}^*)$. Thus, there exists a marking $m'_2 \in \underline{Reach}(flow'_2, \underline{1}^*)$ with two tokens each of which is of the form $\langle p_{in}, 0\rangle$ (Note here that because $flow_2$ is untimed, both tokens have a time-stamp of 0). This mean that $m'_2\,[evt.evt\rangle$ where $evt = \left\langle \underline{t_{flow_1}}, 0, 0 \right\rangle$. In other words, $\underline{t_{flow_1}}$ can fire twice. Clearly, this is a violation of the seriality condition (12). Henceforth, $\underline{serial}(flow_1, flow_2)$ does not hold.

• *only-if part:* Assume that $\underline{serial}(flow_1, flow_2)$ does not hold. Then, there exists $m'_2 \in \underline{Reach}(flow'_2, \underline{1}^*)$, there exists $w'_2 \in \underline{Trace}(flow'_2, \underline{1}^*, m'_2)$, such that the seriablity condition (12) is violated. Without loss of generality, we may assume that the violation occurs for the first time under m'_2. This means that under m'_2, $\underline{p_{flow_1}}$ has one token, say tk', and that one execution of $flow_1$ is already in progress. Note here that tk' is independent of the execution in progress (because $flow_1$ is sound.) Let $\overline{m'}_2$ be the marking at which this execution has started. Since $flow'_2$ is untimed, we can freeze the execution of $flow_1$ and wait for the arrival of tk'. Thus, p_{in} is not 1-safe in $(flow'_2, \underline{1}^*)$, and consequently p_{in} is not 1-safe in $(flow_2, \underline{1}^*)$ (by using (14) from proposition 2.) QED.

To better understand the effect of a structural change on the execution of a procedure, we use the following illustration:

Example 8. Consider the change depicted in figure 8. The sequence

$$sched_{new} = \begin{cases} \langle t_1,0,0\rangle\,\langle t_2,0,4\rangle\,\langle t_3,0,1\rangle & (\beta_0) \\ \quad\langle t_{16},1,0\rangle\,\langle t_{17},1,1\rangle\,\langle t_{18},1,1\rangle\,\langle t_{10},1,1\rangle\,\langle t_{11},2,3\rangle\,\langle t_{19},2,0\rangle & (\beta_1) \\ \quad\langle t_{15},4,1\rangle\,\langle t_8,4,1\rangle\,\langle t_{14},5,0\rangle\,\langle t_9,5,2\rangle\,\langle t_{12},7,0\rangle\,\langle t_{13},7,0\rangle & (\beta_2) \end{cases}$$

is a schedule of $newNet$.
The schedule, $sched_{new}$, consists of three valid sequences; namely β_0, β_1 and β_2. While β_0 is in progress, *newRegion* is inactive. β_2 and β_3 start the moment *newRegion* becomes active. These are *segments* which spawn over *newRegion*, and since $\underline{serial}(newRegion, newNet)$ holds, β_1 and β_2 do not overlap. Thus, *sched* can be viewed as a *segment iteration.* These remarks can be generalized; any schedule of $oldNet$ can be viewed as a segment iteration over *newRegion*. This applies as well to schedules during which do not involve *newRegion* (0 iterations.)

The segment β_1 (resp. β_2) can be decomposed into two independent sequences u_1 and v_1 (resp u_2 and v_2) which reflect the behavior of *newRegion* and the behavior of its context in $newNet$; where:

$$\beta_1 \left\{ \begin{array}{ll} \langle t_{10},1,1\rangle\,\langle t_{11},2,3\rangle & (v_1) \\ \langle t_{16},1,0\rangle\,\langle t_{17},1,1\rangle\,\langle t_{18},1,1\rangle\,\langle t_{19},2,0\rangle & (u_1) \end{array} \right\} \beta_2 \left[\begin{array}{ll} \langle t_8,4,1\rangle\,\langle t_{14},5,0\rangle\,\langle t_9,5,2\rangle\,\langle t_{12},7,0\rangle\,\langle t_{13},7,0\rangle & (v_2) \\ \langle t_{15},4,1\rangle & (u_2) \end{array} \right]$$

To transpose $sched_{new}$ into a schedule of *oldNet*, we will proceed one segment at a time. Consider the following schedules of *oldRegion*:

$$\left\{ \begin{array}{ll} \langle t_6,1,0\rangle\,\langle t_7,1,1\rangle & (\overline{u}_1) \\ \langle 6,4,1\rangle\,\langle t_7,5,0\rangle & (\overline{u}_2) \end{array} \right\} \text{ and note that } \left[\begin{array}{l} \underline{life}\,(u_1) = \underline{life}\,(\overline{u}_1) = [1,2] \\ \underline{life}\,(u_2) = \underline{life}\,(\overline{u}_2) = [4,5] \end{array} \right]$$

First, we *substitute* $\overline{u}_1$ *for* u_1 *in* β_1. Next, we substitute $\overline{u}_2$ for u_2 in β_2. Finally, we combine the resulting sequences with β_0. Note that $\overline{\beta}_1$ and $\overline{\beta}_2$ are segments over *oldRegion*. The sequence

$$sched_{old} = \begin{cases} \langle t_1,0,0\rangle\,\langle t_2,0,4\rangle\,\langle t_3,0,1\rangle & (\overline{\beta}_0) \\ \quad\langle t_6,1,0\rangle\,\langle t_7,1,1\rangle\,\langle t_{10},1,1\rangle\,\langle t_{11},2,3\rangle & (\overline{\beta}_1) \\ \quad\langle t_6,4,1\rangle\,\langle t_8,4,1\rangle\,\langle t_7,5,0\rangle\,\langle t_{14},5,0\rangle\,\langle t_9,5,2\rangle\,\langle t_{12},7,0\rangle\,\langle t_{13},7,0\rangle & (\overline{\beta}_2) \end{cases}$$

is a schedule of $oldNet$. Furthermore, $\underline{life}\,(w_{old}) = \underline{life}\,(w_{new})$.

This substitution mechanism can be used in any structural change as long as the change fulfills the following conditions:

1. *oldRegion* and *newRegion* are sound.
2. *newRegion* $\sqsubseteq_{Time}$ *oldRegion*.
3. $\underline{serial}(oldRegion, oldNet)$.

In this case, any schedule of $sched_{new}$ of *newNet* can be transposed into a schedule of $sched_{old}$ of *oldNet* by segment substitution as outlined above. This will be denoted as $w_{old} \rightsquigarrow_{\delta} w_{new}$.

Can this be generalized to any firing sequence w_{new} of *newNet* which starts at the initial marking? Based on the following case analysis, the answer is in general affirmative.
If w_{new} is a segment iteration over *newRegion*, then we can use the previous segment substitution.
Otherwise, w_{new} can be written as $w_{new} = \alpha.\beta$ where α is a segment iteration over *newRegion* and β is a *partial segment* over *newRegion*. Partial entails that β spawns 1 execution of *newRegion* which is in progress still (i.e. not yet completed.) In our terminology, w_{new} is said to be a *partial segment iteration* over *newRegion*. w_{new} can in general be transposed into a sequence $w_{old} = \overline{\alpha}\overline{\beta}$ where $\alpha \rightsquigarrow_{\delta} \overline{\alpha}$ and $\overline{\beta}$ is a partial segment over *oldRegion* which preserves the contextual behavior of β (i.e. the event firings of w_{new} in the context are untouched.) This assertion fails if *oldRegion* consists of a single transition because $\overline{\beta}$ needs to be a segment. Therefore, we will consider the augmented change for which which the in so-far analysis stands. Further evidence will show that augmenting a change will preserve the context event firings as well as its reachable markings.

Definition 16. Let $flow_1, flow_2 \in \underline{FNets}$ such that:

$$\underline{closed}(flow_1, flow_2) \ \& \ flow_1 \in \underline{SFNets},$$

and let $w \in \underline{Fire}(flow_2, m, m')$ for some $m, m' \in \underline{Mark}(flow_2)$.

- w is a $flow_1$-**segment** iff the following condition holds:

$$v = \underline{project}(w, flow_1) \in \underline{Sched}(flow_1) \ \& \ w[1] = v[1]. \tag{15}$$

Moreover, $\underline{Seg}(flow_1, flow_2)$ denotes the language of all $flow_1$-segments of $flow_2$.

- w is a $flow_1$-**partial segment** iff the following condition holds:

$$v = \underline{project}(w, flow_1) \in (\underline{Exec}(flow_1) - \underline{Sched}(flow_1)) \ \& \ w[1] = v[1]. \tag{16}$$

Moreover, $\underline{PSeg}(flow_1, flow_2)$ denotes the language of all $flow_1$-partial segments of $flow_2$.

- w is a $flow_1$-**segment iteration** iff the following conditions hold:

$$\begin{array}{ll} m \in \underline{1} & \\ \exists \beta_0, \ldots, \beta_n, & \left[\begin{array}{l} w = \beta_0 \ldots \beta_n \\ \forall 1 \leq i < n, \beta_i \in \underline{Seg}(flow_1, flow_2) \\ \beta_0 \notin \underline{VSeq}(flow_1) \ \& \\ \qquad n \neq 0 \Rightarrow \beta_n \in \underline{Seg}(flow_1, flow_2) \end{array}\right. \end{array} \tag{17}$$

Moreover, $\underline{SegIter}(flow_1, flow_2)$ denotes the language of all $flow_1$-segment iterations of $flow_2$.

• w is a $flow_1$-**partial segment iteration** iff the following conditions hold:

$$\exists \alpha \in \underline{SegIter}(flow_1, flow_2),\ \exists \beta \in \underline{PSeg}(flow_1, flow_2),\ w = \alpha\beta \tag{18}$$

Moreover, $\underline{PSegIter}(flow_1, flow_2)$ denotes the language of all $flow_1$-partial segment iterations of $flow_2$.

Note. Note here that the decompositions as described by (17) and (18) are unique because of the extra condition $w[1] = v[1]$. $\underline{Iter}(flow_1, flow_2) = \underline{SegIter}(flow_1, flow_2) \bigcup \underline{PSegIter}(flow_1, flow_2)$. For $w \in \underline{SegIter}(flow_1, flow_2)$, for $0 \leq i \leq n$, β_i is referred to as the i^{th} $flow_1$-iteration of w and is denoted $\underline{iter}(flow_1, w)[i]$. n is called the $flow_1$-iteration degree of w and is denoted $\underline{iter\_deg}(flow_1, w)$. These notions are extended to any $w = \alpha.\beta \in \underline{PSegIter}(flow_1, flow_2)$ as follows:

$$\begin{array}{l}\underline{iter\_deg}(flow_1, w) = \underline{iter\_deg}(flow_1, \alpha) + 1 \\ \underline{iter}(flow_1, w)[i] = \begin{cases} \underline{iter}(flow_1, \alpha)[i] & \text{if } 0 \leq i < \underline{iter\_deg}(flow_1, w) \\ \beta & \text{if } i = \underline{iter\_deg}(flow_1, w) \end{cases}\end{array}$$

Definition 17. Let *repl* be a replacement step such that

$$oldRegion, newRegion \in \underline{SFNets},$$

let $w_1 \in \underline{VSeq}\,(oldNet)$ and let $w_2 \in \underline{VSeq}\,(newNet)$ such that:

$$\underline{project}(w_1, T_{ctxt}) = \underline{project}(w_2, T_{ctxt}),\ where\ T_{ctxt} = \underline{ctxt}(oldRegion, oldNet).tSet,$$

• If $w_1 \in \underline{Seg}(oldRegion, oldNet)$ and $w_2 \in \underline{Seg}(newRegion, newNet)$, then w_1 is a δ-**transposition** of w_2, written $w_1 \leadsto_\delta w_2$, iff the following conditions hold:

$$\underline{life}\,(u_1) = \underline{life}\,(u_2) \tag{19}$$

• If $w_1 \in \underline{PSeg}(oldRegion, oldNet)$ and $w_2 \in \underline{PSeg}(newRegion, newNet)$, then w_1 is a δ-**transposition** of w_2, written $w_1 \leadsto_\delta w_2$, iff the following conditions hold:

$$\begin{array}{r}\exists \bar{u}_1 \in \underline{VSeq}\,(oldRegion), \exists s_1 \in \underline{Sched}^*(flow_1), \\ \exists \bar{u}_2 \in \underline{VSeq}\,(newRegion), \exists s_2 \in \underline{Sched}^*(flow_2)\end{array} \left[\begin{array}{l} u_1.\bar{u}_1 = s_1 \\ u_2.\bar{u}_2 = s_2 \\ \underline{life}\,(s_1) = \underline{life}\,(s_2)\end{array}\right. \tag{20}$$

• If $w_1 \in \underline{Iter}(oldRegion, oldNet)$ and $w_2 \in \underline{Iter}(newRegion, newNet)$, then w_2 is a δ-**transposition** of w_1, written $w_{old} \leadsto_\delta w_{new}$, iff the following conditions hold:

$$\begin{array}{l}\underline{iter\_deg}(oldRegion, w_1) = \underline{iter\_deg}(newRegion, w_2) \\ \underline{iter}(oldRegion, w_1)[0] = \underline{iter}(newRegion, w_2)[0] \\ \forall 1 \leq i \leq \underline{iter\_deg}(oldRegion, w_1), \\ \quad \underline{iter}(oldRegion, w_1)[i] \leadsto_\delta \underline{iter}(newRegion, w_2)[i]\end{array} \tag{21}$$

$$where \begin{cases} T_{old} = oldRegion.tSet\ \&\ T_{new} = newRegion.tSet \\ u_1 = \underline{project}(w_1, T_{old})\ \&\ u_2 = \underline{project}(w_2, T_{new})\end{cases}$$

Note. Note here that as a consequence of (21), w_1 and w_2 are either segment iteration or partial segment iteration.

The next results states that the *transposition operation preserves the life time of a sequence.* Formally,

Lemma 1. *Let repl be a replacement step such that:*

$$oldRegion, newRegion \in \underline{SFNets},$$

let $w_1 \in \underline{Seg}(oldRegion, oldNet)$ *and let* $w_2 \in \underline{Seg}(newRegion, newNet)$.

$$w_1 \rightsquigarrow_\delta w_2 \Rightarrow \underline{life}(w_1) = \underline{life}(w_2)$$

Proof. The proof proceeds by induction on $n = \underline{iter\_deg}(oldRegion, w_1) = \underline{iter\_deg}(newRegion, w_2)$.

- Base case: n = 0. Since $\underline{iter}(oldRegion, w_1)[0] = \underline{iter}(newRegion, w_2)[0]$, the property holds.
- Induction hypothesis: assume that the property holds for all w_1 and w_2 such that $n \leq m$.

Let $w_1 = w_1'.\alpha_1$, $w_2 = w_2'.\beta_2$ such that:

$$\begin{array}{l} \underline{iter\_deg}(oldRegion, w_1') = \underline{iter\_deg}(newRegion, w_2') = m + 1 \\ w_1' \rightsquigarrow_\delta w_2' \\ \beta_1 \in \underline{Seg}(oldRegion, oldNet) \\ \beta_2 \in \underline{Seg}(newRegion, newNet). \end{array}$$

The following property holds by definition:

$$\alpha_1 \rightsquigarrow_\delta \alpha_2 \ \& \ w_1' \rightsquigarrow_\delta w_2'.$$

From the induction hypothesis, we have $\underline{life}(w_1') = \underline{life}(w_2')$.
From the definition of segment replacement, we have $\underline{life}(\alpha_1) = \underline{life}(\alpha_2)$.
Therefore, $\underline{life}(w_1) = \underline{life}(w_2)$. QED.

The next result relates the seriality and the segment iteration properties. In essence, it states that *seriality of sound flow net is equivalent to its iteration.* Formally,

Lemma 2. *Let* $flow_1, flow_2 \in \underline{FNets}$ *such that:*

$$\underline{closed}(flow_1, flow_2) \ \& \ flow_1 \in \underline{SFNets}.$$

$\underline{serial}(flow_1, flow_2)$ *iff the following condition holds:*

$$\underline{Exec}(flow_2') \subseteq \underline{Iter}(flow_1', flow_2'). \tag{22}$$

$$where\ flow_1' = \underline{aug}(flow_1) \ \& \ flow_2' = \underline{sep}(flow_1, flow_2).$$

Proof. Note here that (22) $\Leftrightarrow$ (12).

Extensions. (Priorities, Generalized Time Model)

1. An extension of the model through priorities requires that all the input transitions of a closed sound subnet $flow_1$ to have the same priority, say $x \in \mathbb{N}$. Otherwise, $flow_1$ would have a dead (input) transition. By setting the priority of $\underline{t_{flow_1}}$ to x, propositions [2,3,4], and lemmas [1, 2] hold.
2. These results hold regardless of the way events firing delays are computed.

2.3 Static Model Properties

The following result examines in more detail the effect that a structural change has on the semantics of a flow net. It tells us that whenever its conditions are met by a change, 1- *the behavior of its context is preserved* and 2- *any execution of the new net can be transposed into an execution of the old net.* This means, that properties such as the different approximation properties and the seriality property introduced earlier are preserved by the change.

Proposition 5. *Let repl be a replacement step such that the following conditions holds:*

$$\begin{array}{l} oldRegion, newRegion \in \underline{SFNets} \\ \underline{serial}(oldRegion, oldNet) \\ newRegion \sqsubseteq_{Time} oldRegion. \end{array}$$

The following property holds:

$$\forall w_2 \in \underline{Exec}(newNet') \begin{bmatrix} (Q_1) & w_2 \in \underline{Iter}(newRegion', newNet') \\ (Q_2) & \exists w_1 \in \underline{Exec}(oldNet'), \begin{bmatrix} w_1 \leadsto_{\delta'} w_2. \\ m_1{\downarrow}_{Ctxt} = m_2{\downarrow}_{Ctxt}. \end{bmatrix} \end{bmatrix} \quad (23)$$

$$where\ repl' = \underline{aug}(repl)\ \&\ Ctxt = \underline{ctxt}(repl)\ \&\ \underline{1}^* [w_1\rangle\, m_1\ \&\ \underline{1}^* [w_2\rangle\, m_2$$

Proof. The proof is carried out by induction on w_2.

- *base case*: $w_2 = \lambda$. In this case $w_2 = w_1$ and $m_1 = m_2 = \underline{1}^*$ satisfy Q_1 and Q_2.
- *induction hypothesis*: Assume that Q_1 and Q_2 are met by w_1, m_1, w_2 and m_2.
- *induction*: Let $evt_2 \in \mathcal{E}newNet'$ such that:

$$m_2 [evt_2\rangle\, \overline{m}_2\ \&\ evt_2 = \langle t_2, en\_time_2, fr\_delay_2 \rangle .$$

<u>First case</u>: $t_2 \in Ctxt.tSet$; in other words, evt_2 is a context event.

Since m_1 and m_2 are context equivalent (by the induction hypothesis), evt_2 is enabled under m_1 (but not necessarily time enabled), thus $m_1 [[evt_1\rangle$ holds. We shall consider the following two (sub-)cases:

1. evt_2 is time enabled under m_1. Thus, there exists a marking $\overline{m}_1$ such that $m_1 [evt_2\rangle\, \overline{m}_1$. In this case, Q_1 and Q_2 are met by $w_1.evt_1$, $\overline{m}_1$, $w_2.evt_1$ and $\overline{m}_2$.
2. evt_2 is not time enabled under m_1. This necessarily, implies that all events which are time enabled under m_1 are local to *oldRegion'*. Thus, *oldRegion'* is active under m_1, which necessarily implies that *newRegion'* is also active under m_2. This is a consequence of $w_1 \leadsto_{\delta'} w_2$ of the induction hypothesis. Therefore, w_1 and w_2 are partial segment iterations.

   w_1 can be written as $w_1 = \alpha_1.\beta_1$ and w_2 can be written as $w_2 = \alpha_2.\beta_2$, where:

$$\begin{bmatrix} \alpha_1 \in \underline{SegIter}(oldRegion', oldNet') \\ \beta_1 \in \underline{PSeg}(oldRegion', oldNet') \end{bmatrix} \begin{bmatrix} \alpha_2 \in \underline{SegIter}(newRegion', newNet') \\ \beta_2 \in \underline{PSeg}(newRegion', newNet') \end{bmatrix} \begin{bmatrix} \alpha_1 \leadsto_{\delta'} \alpha_2 \\ \beta_1 \leadsto_{\delta'} \beta_2 (*) \end{bmatrix}$$

   Let $u_1 = \underline{project}(\beta_1, T_1)$ and let $u_2 = \underline{project}(\beta_2, T_2)$. From (*), the following property holds:

$$\begin{array}{r} \exists \overline{u}_1 \in \underline{VSeq}\,(oldRegion), \exists s_1 \in \underline{Sched}(flow_1), \\ \exists \overline{u}_2 \in \underline{VSeq}\,(newRegion), \exists s_2 \in \underline{Sched}(flow_2) \end{array} \begin{bmatrix} u_1.\overline{u}_1 = s_1 \\ u_2.\overline{u}_2 = s_2 \\ \underline{life}\,(s_1) = \underline{life}\,(s_2) \end{bmatrix}$$

   Let k be such that $s_1[k-1].en\_time < en\_time_2 \le s_1[k].en\_time$, and let β_1' be such that

$$\underline{project}(\beta_1', T_{ctxt}) = \underline{project}(\beta_1, T_{ctxt})\ \&\ \underline{project}(\beta_1', T_1) = s_1[1] \dots s_1[k].$$

By construction, $\beta_1' \rightsquigarrow_{\delta'} \beta_2$ holds. Thus, the sequence $w_1' = \alpha_1.\beta_1'$, the marking m_1' such that $\underline{1}^* [w_1'\rangle\, m_1'$, w_2 and m_2 satisfy the conditions Q_1 and Q_2. Moreover, $m_1' [evt_2\rangle$ holds. We are back to the previous case.

Second case: $t_2 = \underline{t_{newRegion'}}$; in other words, *newRegion'* is activated under m_2.

w_1 and w_2 are necessarily segment iterations. Thus, $m_1 = m_2$, and consequently *newRegion'* can also be activated under m_1. Let $evt_1 = \left\langle t_{oldRegion'}, en\_time_2, 0\right\rangle$. Then, $m_1 [evt_1\rangle\, \overline{m}_1$ holds for some marking $\overline{m}_1$. From *newRegion* $\sqsubseteq_{Time}$ *oldRegion*, we can conclude that $evt_1 \rightsquigarrow_{\delta'} evt_2$. Thus, $w_2.evt_2$, $\overline{m}_2$, $w_1.evt_1$, $\overline{m}_1$ meet Q_1 and Q_2. *Case Closed.*

Third case: $t_2 = \overline{t_{newRegion'}}$; in other words, *newRegion'* produces a token.

w_1 and w_2 are both partial segment iterations, and therefore they can be decomposed as outlined in the first case. Thus, $m_1 [\overline{u}_1\rangle\, \overline{m}_1$ for some marking $\overline{m}_1$. Moreover, $w_2.evt_2$, $\overline{m}_2$, $evt_1.\overline{u}_1$, $\overline{m}_1$ satisfy Q_1 and Q_2.

Fourth case: $t_2 \in newRegion.tSet$; in other words evt_2 is a local event of *oldRegion*.

w_1 and w_2 are both partial segment iterations, and therefore they can be decomposed as outlined in the first case. Since *newRegion'* is also sound, the execution $u_2.evt_2$ can be prolonged into a schedule s_2' of *newRegion'*. This, means that there exists a schedule s_1' of *oldRegion'* such that $\underline{life}(s_1') = \underline{life}(s_2')$. The idea here is to undo all the event firings of u_1 in w_1, to replace them by event firings from s_1' and to make sure that the context is not disturbed by these operations.

Let k be such that $s_1'[k-1] \leq \tau < s_1'[k]$ where:

$$\tau = min\left\{evt.en\_time \mid m_2 [evt\rangle \ \&\ evt.tr \in T_{ctxt}\right\},$$

and let β_1' be such that

$$\underline{project}(\beta_1', T_{ctxt}) = \underline{project}(\beta_1, T_{ctxt}) \ \&\ \underline{project}(\beta_1', T_1) = s_1'[1] \ldots s_1'[k].$$

By construction, $\beta_1' \rightsquigarrow_{\delta'} \beta_2$ holds. Thus, the sequence $w_1' = \alpha_1.\beta_1'$, the marking m_1' such that $\underline{1}^* [w_1'\rangle\, m_1'$, $w_2.evt_2$ and $\overline{m}_2$ satisfy the conditions Q_1 and Q_2. *QED*

Theorem 1. *Let repl* $= \langle oldNet, \delta, newNet\rangle$ *be a replacement such that:*

$$\begin{array}{l} oldRegion, newRegion \in \underline{SFNets} \\ \underline{serial}(oldRegion, newRegion) \\ newRegion \sqsubseteq_{Time} oldRegion \end{array}$$

The following properties hold:

1. $\underline{serial}(newRegion, newNet)$.
2. $newNet \sqsubseteq_{Time} oldNet$.
3. $newRegion \sqsubseteq_{Sched} oldRegion \Rightarrow newNet \sqsubseteq_{Sched} oldNet$.
4. $oldNet \in \underline{SFNets}\ \&\ oldRegion \sqsubseteq_{Time} newRegion \Rightarrow newNet \in \underline{SFNets}$.

Proof. Let $repl' = \underline{aug}(repl)$, let $Ctxt = \underline{ctxt}(repl)$, let $T_{ctxt} = Ctxt.tSet$ and let $P_{ctxt} = Ctxt.pSet$.

The properties 1, 2 and 3 are direct consequences of proposition 5. By using proposition 3, it is clear that we only need to prove that *newNet'* is sound starting from the premise that *oldNet'* is sound.

• First, we prove that *newNet'* does not have a dead transition. Let t be a transition of *newNet*. Consider the following cases:

1. $t \in T_{ctxt}$; where $T_{ctxt} = \underline{ctxt}(repl).tSet$.
   - There exists a marking $m_1 \in \underline{Reach}(oldNet', \underline{1}^*)$, there exists an execution $w_1 \in \underline{Exec}(oldNet')$, there exists an event evt_1 of $oldNet'$ such that:

$$evt_1.tr = t \; \& \; \underline{1}^* \, [w_1\rangle \, m_1' \, [evt_1\rangle \, m_1.$$

   - Since $oldRegion \sqsubseteq_{Time} newRegion$, the results from proposition 5 can be applied to the replacement step $repl'' = \underline{aug}(repl)^{-1}$. Thus, there exists a marking $m_2 \in \underline{Reach}(newNet', \underline{1}^*)$, there exists $w_2 \in \underline{Fire}(\underline{1}^*, m_2,)$, such that:

$$m_1{\downarrow}_{P_{ctxt}} = m_2{\downarrow}_{P_{ctxt}} \; \& \; w_2 \rightsquigarrow_{repl''} w_1.evt_1.$$

   This implies that evt_1 is an element of w_2. In other words, t is not dead in $newNet'$.
2. $t \in newRegion'.tSet$.
   - Since $oldNet'$ is sound, there exists $m_1 \in \underline{Reach}(oldNet', \underline{1}^*)$ under which a token tk is about to be consumed by $oldRegion'$. This implies that under m_1, $oldRegion'$ is not active. In other words, $m_1 = m_1{\downarrow}_{P_{ctxt}}$.
   - Since $oldRegion \sqsubseteq_{Time} newRegion$, the results from proposition 5 can be applied to the replacement step $repl'' = \underline{aug}(repl)^{-1}$. Thus, $m_1 \in \underline{Reach}(newNet, \underline{1}^*)$. Hence, tk is allowed in $newRegion'$. Since $newRegion'$ is also sound, there exists $s_2 \in \underline{Exec}(newRegion')$ that $\underline{start}\,(s_2) = tk.av\_time$ and the last element of s_2 is of the form $\langle t, \tau\rangle$. In other words, t is locally "not dead" in $newRegion'$. All the event firings of s_2 take place in $newNet'$ starting from m_1. These event firings are separated by context event firings. Thus, t is not dead in $newNet'$.

• Next, we prove that $newNet'$ has the well termination property. Let $m_2 \in \underline{Reach}(newNet', \underline{1}^*)$. Consider the following cases:

(a) $m_2 = m_2{\downarrow}_{P_{ctxt}}$; in other words, $newRegion'$ is not active. Thus, $m_2 \in \underline{Reach}(oldNet', \underline{1}^*)$. From the soundness of $oldNet'$, we have that a terminal marking m_2' is reachable from m_2 in $oldNet'$. This means that the same m_2' is reachable from m_2 in $newNet'$.

(b) $m_2 \neq m_2{\downarrow}_{P_{ctxt}}$; in other words, $newRegion'$ is active. Note here that it is always possible to flush the tokens out of $oldRegion'$. Thus, there exists a marking $m_2'' \in \underline{Reach}(newNet', m_2)$, such that $m_2'' = m_2''{\downarrow}_{P_{ctxt}}$. We are back to the first case.

Extensions. (Priorities, Generalized Time Model)

1. Proposition 5 and theorem 1 hold in a model extended with priorities, if we add the requirement that the input transitions of *oldRegion* and the input transitions of *newRegion* to have the same priority.
2. These results hold regardless of the way events firing delays are computed.

2.4 Sound Transformations

Next, we enumerate some transformations which preserve the soundness property. The reader is referred to figure 9 for illustration. These reversible transformations have been introduced and investigated by Van der Aalst in [20] for (untimed) workflow nets.

- **T1** *division*: An activity t is divided into two consecutive activities t_1 and t_2 which are such that

$$delay(t_1) \oplus delay(t_2) = delay(t).$$

- **T2** *specialization*: An activity t is replaced by two conditional activities t_1 and t_2 which are such that

$$delay(t_1) \bigcup delay(t_2) \in \mathbb{T}$$
$$delay(t_1) \bigcup f delay(t) = delay(t).$$

- **T3** : An activity t is replaced by two parallel activities t_1 t_2 which are such that

$$delay(t_1) \otimes delay(t_2) = delay(t).$$

- **T4** : An immediate activity t_1 is replaced by the iteration of an immediate activity $t2$.
- **T5** *expansion* an activity t is replaced by a sound flow net *flow* such that:

$$\underline{cpl\_time}(flow) = delay(t).$$

Other rules such as sequentialization, parallelization and swapping can be derived from the previous ones.

Proposition 6. *The reversible transformations T1-T5 perserve soundness and time approximation.*

Proof. Note here that any replacement step *repl* which makes use of these transformations verify the conditions of theorem 1. The result follows from the property 3 of the very same theorem.

3 Process Structural Changes: The Dynamic Model

The introduction of a structural change into a workflow procedure, if not carefully *managed*, may be error-prone, and could lead to inefficiencies or system breakdowns. There is a large myriad of issues to be addressed within the context of change management (these issues will be discussed in a forthcoming work;) including:

scoping: The issue here is to devise strategies to determine the jobs of the procedure which are affected by the change.

Scoping refers to the ability to identify the jobs which may be affected by the change.

In some situations, it may be desirable to scope in the jobs which have already completed (e.g. recalling defected parts.) In other situations, it may be necessary to scope out the running jobs which have already gone through a specific change, state or what have you (e.g. irreconcilable changes, or the change is harmful after the state is reached.) Traditionally, only the running jobs have been considered by the different models proposed in the literature, and the tradition lives on with this work. The *scope* of a change consists of all the jobs which may be affected by the change.

migration: In order for the change to take place, the in-scope jobs need to go through a migration phase.

Migration refers to the ability to define how the in-scope jobs evolve.

Previously, in [9] the authors have considered the following *migration policies*:
Resubmit: all the job's work units are canceled first, then the job is resubmitted to the new procedure for processing. This policy, generally *safe*, may be recommended to fix *hard bugs* or to conform to *hard regulations.* In some cases, it may not be cost effective; hours of work and almost finished jobs are lost.

Wait : The change does not proceed until the job reaches a safe state. This solution may not be feasible; it may take some time before the system reaches the sought state. It is also inadequate to deal with punctual changes such as exceptions.

Transfer: Work units associated with the job are transfered to the new procedure. The transfer may affect all or some of the work units. The work units not affected by the transfer continue their progression in the old net as if *the change never took place.* The transfer can also be optimized to ensure safeness. It is based upon the principle of change locality; the units of work evolving outside of the old change region remain unchanged in the new procedure. The work units bordering the old change region are moved to the interface of the new region. Some of the work units progressing inside of the old change region are moved to the new change region, others terminate their progression in the old change region and are moved to the new region when they reach the exit place. Additionally, the connection to the old change region's entry place is severed so as not to allow in any new work unit. In [9], the authors have introduced the *Synthetic Cut-Over Change (SCOC)* to reflect the situation where no work unit can be safely transfered from the old change region. Heuristics have also been devised to determine cases where SCOC is a safe solution. The work has been expanded in an in-progress work [13] through the *Extended Synthetic Cut-Over Change (E-SCOC)* which reflects the case where the token transfer is partial.

3.1 Timed Adaptive Flow Nets

The transfer policy has a couple of shotcomings; (1) its restraining nature, in the sense that non-transfered tokens are "captive" within the old region; the only way out is the exit place of the old change region (2) its lack of activity cancellation support. An improvement is to provide support for progessive transfer (instead of an abrupt one). Tokens are maintained in the old net, but can jump into the new net using *jumpers.* A jumper is a high-level box which has a non empty set of *inlets*, a (possibly empty set) of *outlets* and a *set up time stamp.* The inlets (resp. outlets, are connecting points in the old (resp. new) net, the set up time stamp reflects the time at which a jumper is set up. A jumper is functionally similar to a silent instantaneous transition. It can be classified according to its configuration as follows:

1. A *mover* is a jumper with at least one outlet. It is can be used to move a job's work units from one procedure to another.
2. An *annealer* is a jumper with no outlets. It is used, typically, to flush a job's work units.

Definition 18. Let $net_1, net_2 \in \underline{Nets}$.

A (net_1, net_2)-**jumper** is a system, $jr = \langle inLet, outLet, up\_time \rangle$, which consists of:

- $inLet \subseteq net_1.pSet$, the (non empty) set of *inlets* of jr.
- $outLet \subseteq net_2.pSet$, the set of *outlets* of jr, which are such that $inLet \bigcap outLet = \emptyset$.
- $up\_time \in \mathbb{Q}^+$, the *set up time* of jr.

Moreover, $\underline{Jumpers}(net_1, net_2)$ denotes the class of all (net_1, net_2)-jumpers.

Extensions. Our model of jumpers can be easily extended to accommodate jumpers with delays, labels, priorities, etc... For instance, on can define the priority of a jumper to be higher than the priority to any activity to support a *must-jump* semantics. Note here that if net_1 and net_2 coincide (which by the way is not prevented by the definition), then a jumper can be used to modify the execution within a net. Thus, the jumpers model **includes** support for **redo**, **undo** and **goto**.

Example 9. Consider the change in the order processing procedure introduced in example 6. For this particular change, it has been decided to apply a common migration policy to all running jobs. This policy calls for an expeditive migration (see figure 10.) which states that:

1. every order which has just came out of the approval should be sent to the new billing activity (jumper jmp_8.)
2. every order which has been routed to *billing* and *shipping* should be sent to the new billing activity (jumper jmp_9.)
3. every order which has been billed but not shipped should be sent to the new shipping activity (jumper jmp_{10}.)

The setup time of all these jumpers is 8:30 a.m.

After the jumpers are set up, the execution of the job resumes in a *timed adaptive flow net.* Each timed adaptive flow net has a sequence of *versions* and a set of *jumpers.* Each version is a timed flow net which represents a version of the procedure which results from a previously carried out structural change. Although Token jumping is restricted from a one version to its successor, this restriction is not essential to the model. The last element in the sequence (i.e. the last version) is referred to as *the root.* Formally,

Definition 19. A **timed adaptive net** is a system, $net = \langle verSeq, jSet \rangle$, which consists of:

- a non empty sequence, $verSeq$, of timed nets, referred to as *versions.*
- a finite set, $jSet \subset \underline{Jumpers}$, of *jumpers*

which are such that

$$\begin{array}{l}\forall jr \in jSet,\ \exists\, 1 \leq i < n,\ jr \in \underline{Jumpers}(verSeq[i], verSeq[i+1]) \\ \forall\, 1 \leq i < n,\ \forall jr_1, jr_2 \in (jSet \bigcap \underline{Jumpers}(verSeq[i], verSeq[i+1])),\ jr_1.up\_time = jr_2.up\_time\end{array}$$

$$\text{where } n = \underline{lgth}(versions)$$

Moreover, $\underline{ANets}$ denotes the class of all timed adaptive nets.

A **timed adaptive flow net** is a timed adaptive net, *flow*, which is such that

$$flow.verSeq \in \underline{AFlowNets}^{+}.$$

Moreover, $\underline{AFlowNets}$ denotes the class of all timed adaptive flow nets.

Note. The last version of *net* is referred to as the *root* of *net*, and will be denoted $\underline{root}(net)$. The notion of element carries over to to timed adaptive nets through their sequence of versions. The *versioning* associated with *net*, is the function, $\underline{version}_{net} : \underline{Elem}(net) \bigcup jSet \longrightarrow \mathbb{N}$, defined as follows:

$$\begin{array}{l}\forall x \in \underline{Elem}(net),\ \underline{version}_{net}(x) = i \Leftrightarrow x \in verSeq[i]. \\ \forall jr \in jSet,\ \underline{version}_{net}(jr) = i \Leftrightarrow jr \in \underline{Jumpers}(verSeq[i], verSeq[i+1]).\end{array}$$

Extensions. A possible extension is to relax the linear ordering of versions into a partial ordering. This, will allow not only **branched**, instead of linear, **versioning**, but also a change to be carried out on any early version. In this case, versions can be labeled as 1.0, 1.1 etc...based on their topological ordering, and any maximal version (w.r.t. partial order) can be considered as a root. We will sketch later how the dynamic change model can easily accommodate these extensions.

Example 10. Figure 10 shows a timed adaptive flow net, *AOrder*, for the order processing procedure. *AOrder* has two versions, $OrderProc_1$, and $OrderProc_2$ (only one change has been carried out thus far.)

In the context of this work, the dynamic structural model is defined in terms of *dynamic change instance.* The instance description includes the replacement step (to describe the structural change,) the jumpers set up, the old and the new nets and the change enabling time (to reflect the time at which the change instance is enacted.) Formally,

Definition 20. A **dynamic replacement instance** is a system, $inst = \langle oldNet, newNet, \delta, jSet\rangle$, which consists of:

- $\delta \in \underline{ReplPairs}$, the *replacement pair* of *inst.*
- $jSet \in \underline{Jumpers}$, the *jumpers set up* of *inst.*
- $oldNet \in \underline{ANets}$, the *old net* of *inst.*
- $newNet \in \underline{ANets}$, the *new net* of *inst.*
- $en\_time \in \mathbb{Q}^{+}$, the *enabling time* of *inst.*

which are such that

$$\begin{array}{l} oldRoot \rightsquigarrow_{\delta} newRoot \\ \forall jr \in jSet,\ \left[\begin{array}{l} jr.inLet \subseteq oldRoot \\ jr.outLet \subseteq newRoot \\ jr.up\_time = en\_time \end{array}\right. \end{array} \tag{24}$$

$$\text{where } oldRoot = \underline{root}(oldNet),\ newRoot = \underline{root}(newNet).$$

Moreover, $\underline{DynChgs}$ denotes the class of all dynamic change instances.

Note. Conditions(24) ensure that (1) the change is carried out on the last version of a timed adaptive net, (2) token jumpings occur from the last version to the new version and (3) the jumpers are set up the moment the change instance is enabled.
For $net \in \underline{ANets}$, $\underline{DynChgs}(net) = \{chg \in \underline{DynChgs} | chg.oldNet = net\}$.
To support branched changes or versioning, the above definition should be changed by allowing $oldRoot \in \underline{root}(oldNet)$.

All the management policies, described earlier, can be modeled through jumpers. The SCOC can be implemented by setting context tokens movers. Additionally, the E-SCOC uses extra movers to move tokens from within the old change region inside of the new change region. The Wait policy setting up jumpers from a the old net's safe state to a safe state in the new net. The resubmit policy can be implemented using annealer to flush all work units but one that is moved to the entry of the new procedure.

3.2 Semantics of Timed Adaptive Flow Nets.

In the absence of a rigorous semantics, timed adaptive flow nets are with no foundation. In what follows, we propose to treat transition firings, token jumps, and dynamic changes as events, that occur as a timed adaptive flow net evolves structurally and stately. The result is a *uniform*, *eager firing*, and *FIFO* semantics which in addition supports **change composition**. To formalize this semantics, we need to adopt the following conventional notations:

Note. The notions of marking carries over to timed adaptive nets through their versions. For $flow \in \underline{AFlowNets}$, the entry place of the first version of $flow$ will be also considered as the entry place of $flow$, the notion of initial marking carries over to $flow$ through its first version. The exit place of the root of $flow$ will be considered as the exit place of $flow$, and thus the notion of terminal marking will carry over to $flow$ through its last version.

Note. For $net \in \underline{ANets}$ and $jr \in net.jSet$, jr will be considered as a silent instantaneous transition, t_{jr}, whose input set is $jr.inLet$, and whose output set is $jr.outLet$. Thus a tokens jump in net, is a special event of $net^{\oplus}$ (the timed net obtained from net by assimilating jumpers to transitions.) What is special about a tokens jump is that it can occur only after the jumper is set up. Fortunately, this can be easily accommodated by adding the following condition to the conditions 4 of definition 5:

$$jr.up\_time \leq evt.en\_time \text{ if } evt.tr = t_{jr} \text{ for some } jr \in jSet. \tag{25}$$

An element of $\underline{Evts}\,(net) = \underline{DynChgs}(net) \bigcup (\underline{Evts}\,(net^{\oplus}))$ will be referred to as an **event** of net.
By considering a jumper as a transition, the reader can see that the jumpers extensions discussed earlier can, indeed, be easily incorporated into our model.

First, we include the dynamic change model in the semantics as follows:

Definition 21. Let $mnet = \langle net; m\rangle$ be a marked timed adaptive net, and let $chg \in \underline{DynChgsnet}$.

chg is **time enabled** in $mnet$, written $mnet\,[chg\rangle$, iff

$$\begin{array}{l} \forall tk \in m,\ m.en\_time \leq chg.en\_time. \\ net = chg.oldNet \end{array}$$

In this case, the **time enabling** of chg in $mnet$ leads to the marked timed adaptive net, $mnet' = \langle net'; m'\rangle$, written $mnet\,[chg\rangle\, mnet'$, where $(net' = chg.newNet \;\&\; m = m')$.

Second, tokens jumps and events firings within a version are included in the semantics as follows:

Definition 22. Let $mnet = \langle net; m\rangle$ be a marked timed adaptive net, and let $evt \in \underline{Evts}\,(net^{\oplus})$.

evt is **time enabled** in $mnet$, written $mnet\,[evt\rangle$, iff $m\,[evt\rangle_{net^{\oplus}}$.

In this case, the **time enabling** of evt in $mnet$, leads to the marked timed adaptive net $mnet' = \langle net'; m'\rangle$, written $mnet\,[evt\rangle\, mnet'$, where:

$$net' = net \;\&\; m\,[evt\rangle_{net^{\oplus}}\, m'.$$

Note. As usual, we will consider the transitive closure of the "time enabling" relation. Thus, for $mnet = \langle net, m\rangle$ and $mnet' = \langle net'; m'\rangle$ marked timed adaptive flow nets, and $w \in \underline{Evts}\,(net')$, such that $mnet\,[w\rangle\, mnet'$, w is called an *adaptive event sequence* leading from $mnet$ to $mnet'$. $\underline{AFire}(mnet, mnet')$ denotes the language of all such adaptive firing sequences, this language will be denoted $\underline{AFire}(net, m, m')$ if $net = net'$. Note here the **information completness** nature of an adaptive firing sequence; in the sense that it gives a full account of (1) the activity firings, (2) the structural changes, and (3) for each enabled change, its start time (i.e. when the first jump to the new net occurs) and its end time (i.e. when all previous versions are inactive.) The forgetful operator $\oplus$ allows us to forget about the structural changes, thus $w^{\oplus}$ is the sequence obtained from w by erasing the dynamic change instances. All the notions introduced for flow nets will expand to adaptive flow nets. The job model remains essentially the same (except for the history).

Definition 23. Let $flow \in \underline{AFlowNets}$.

A **job** over $flow$ is a system, $job = \langle name, state_{in}, state_{cr}, hist\rangle$, which consists of:

- $name \in \mathcal{JN}$, the names of the job.
- $state_{in} \in \underline{1}_{flow.verSeq[1]}$, the initial state of the job.
- $state_{cr} \in \underline{Mark}(flow)$, the current state of the job.
- $hist \in \underline{AFire}(flow, state_{in}, state_{cr})$, the history of the job.

3.3 Case Study

The initial story...

For our case study, we have chosen a business process within a fictitious hotel chain called *The Desert Inn* (the initial story has been inspired from the running example of Bichler et al. [28].) This process handles many of the hotel activities including reservation, billing, check in and check out. The business model of *The Desert Inn* is based on *service differentiation*; there are two kinds of customers, namely *Highly Important Customers (HIC)s* and *Very Important Customers (VIC)s* (usually, everybody gets a high designation.)

In a typical situation, a customer, either in person (or through a travel agent,) makes a reservation by calling the toll-free number 1-800-DESERT-IN of the hotel reception desk. A host greets the customer, collects the necessary information, and processes the customer request while the caller is waiting. The request is either rejected or confirmed. In the latter case, a confirmation number is issued to the customer. In some cases, the requests are processed off-line during the next business day.
A customer may at any time cancel a reservation; the *Desert Inn* waives the cancellation fee for *HIC*s, but usually charges *VIC*s the equivalent of one night stay. In the latter case, a bill is sent to the customer, a 2-week payment due reminder is set up and if a payment is not received within the 2-month delay, the case is handed over to the collection agency for legal actions.

A *VIC* pays the bill before checking out, whereas a *HIC* pays after receiving the bill.

The initial version of this process, referred to as $DIbp_1$, is depicted in figure 11.

When a customer call is routed to a live agent, it is the beginning of a case. The live agent greets the customer and collects the necessary information (activity *Greet_Customer*), and flags the case as either *HIC* or *VIC*. In the former case, the request is processed while the customer is on line and a confirmation number is issued to the customer (activity *Online_Proc.*) Here, we assume that a *HIC* case is very seldom rejected. In the latter case, the request is put in the queue of *pending requests* to be processed during the next business day (activity *Batch_Proc.*) A pending request is either rejected (activity *Reject,*) and the case terminates, or confirmed and a confirmation letter is fax-ed to the customer (activity *Confirm.*)

When a customer (with a confirmed reservation) is about to check in, the state of his case is either $\{p_5, p_{12}\}$ (if he/she is a *HIC,*) or $\{p_5, p_8\}$ (if he/she is a *VIC.*) In the former case, the customer checks in (activity *Check_In*), gets his bill and pays for it (activity*Bill & Pay*), checks out (activity *Check_Out,*) then the case is archived (activity *Archive.*) In the latter case, The customer checks in and checks out, then he/she is billed (activity *Bill_Customer,*) and when the customer pays his/her bill (activity *Pay_Bill,*) the case is archived.

The cancellation of a reservation proceeds by firing *Cancel_Res*, followed by the firing of either *Waive_Fee* (for a *HIC*) or *Charge_Fee* (for a *VIC.*)

And the troubles begin...

On one hand, a wave of reservation requests, reservation losses and cancellations hits *The Desert Inn.* The staff is overwhelmed by the volume of telephone calls they have to deal with, causing considerable confirmation delays and missed business opportunities. Dissatisfied customers and travel agencies cancel their reservations and refuse to pay the cancellation fee. On the other hand, a serious bug in the billing system raises some concerns among the IT department of the hotel.

The solution...

To respond, the management staff decides to take the following measures:

- Deploying a series of upgraded systems including *EZ_Bill* for billing, *EZ_Pay* for payment processing, *EZ_Res* for reservation online processing, *EZ_In* for speedier check in, *EZ_Out* for smooth check out, and *EZ_Cancel* to deal with the cancellation process. *EZ_Res* is a secured online fully automated reservation system with enhanced voice coaching capabilities.
- All *VIC* reservation requests are to be handled by *EZ_Res.*
- Every *VIC* must pay a deposit, the equivalent of one night stay, in advance (i.e. prior to check in.) This transaction is conducted on-line by the *EZ_Res* system. A confirmation message (with a reservation number and a credit card approval number,) or a rejection message, is either voice-read (if the customer remains on line,) or fax-ed (if the customer chooses so.)
- A pro-rated cancellation refund policy is applied to all canceled reservations.
- Unless an advance delay notification is received by the reception staff, the latest guaranteed check in time is 3.00 p.m.
- A highly trained team of agents, *The Desert Inn Club* team, is dedicated to handle the *HIC*s.
- The collection agency is excused.
- Every customer pays his/her hotel bill before he/she can proceed to check out.
- A new discounted plan, *The Desert Inn Passport (DIP)*, with some restrictions including 6-week advanced booking-and-payment and 72-hour advanced-cancellation restrictions, is to be introduced within 6 weeks.

All new cases are to be handled in accordance with these measures. The new version of this process, referred to as $DIbp_2$, is depicted in figure 12. The automated activity *EZ_Greet* greets the customer, prompts and collects some preliminary information. The *DI_Club* activity models the processing of a *HIC* case by a member of *The Desert Inn Club* team. The *No_Show* activity is fired when a customer fails to check in before 3.00 p.m.

Not so fast, what about the current cases?...

These measures are critical and therefore effective immediately. The migration policy is as follows:

1. Send a policy change notification letter, by fax, to all *VIC*s wih a pending request. The letter should bring their attention to pay a deposit within 24 hours, otherwise their requests will be rejected. To deal with the expected wave of responses, a light version of *EZ_Res*, *Express_Res*, is deployed.
2. Send a similar letter to all confirmed *VIC*s who have not yet checked in. The letter accords a delay of one week. The expected call volume will be handled by a temporary automated activity called *Pay_Deposit*.
3. All confirmed and not yet checked in *HIC*s will be sent to *The Desert Inn Club* team for processing.
4. All checked in customers who have not yet checked out should be billed correctly. Two temporary activities, *Correct_Bill* and *Correct_Payment* are available.
5. Waive the cancellation fee for all currently canceled customers.
6. Do not apply the pay-before-you-leave policy to the currently checked in *HIC*s.

The implementation...

In order to comply with the migration policy, we will use **macros**. A macro is a jumper with an associated sound timed flow net (or activity,) referred to as its **bridge**. The execution of a macro proceeds as follows:

1. a token is selected from each inlet using a *FIFO* policy.
2. the selected tokens are fused.
3. the resulting token is injected into the bridge.
4. the bridge is executed (by virtue of the soundness, the termination is guaranteed.)
5. the produced token is split into several tokens, one of which is deposited into each outlet (if the macro has no outlets, then the produced token *vanishes*.)

Macros are very powerful; they can be used to implement corrective measures, ad-hoc migration, and even negotiation (this latter point will be discussed in length in a forthcoming paper.)

Note. All the macros and jumpers discussed below are depicted in figure 13.

• The first measure is implemented through the macro $macro_1$; the activity *Set_Reminder* is used to set up a timer; the firing of the activity *timer_expired* signals the expiration of the timer.

• The second measure is implemented through $macro_2$.

• The third measure is implemented through jmp_1.

• The fourth measure is implemented on a case basis as follows:

1. jmp_2 moves the *VIC* cases which have not been billed to the *EZ_Bill* activity in the new version.
2. $macro_3$ handles the *VIC*s who have already payed their bills, but have not yet checked out.
3. $macro_4$ handles the *HIC*s who have not yet been billed.
4. $macro_5$ handles the *HIC*s who have already been billed, but have not yet paid.

5. $macro_6$ handles the *HIC*s who have already paid their bills

• The fifth measure is implemented through jmp_3 to waive the cancellation fee for *VIC*s, and jmp_4 to waive the cancellation fee for *HIC*s.

• No special arrangement is necessary to implement the sixth measure. The migration will take place after the customer checks out (see $macro_4$.)

4 Related Work

Recently, the problem of workflow structural change has been the focus of numerous work efforts, but none of these efforts consider the time issues. Thus, our comparison will be done with respect to the untimed adaptive flow nets.

In [4], the authors introduce a class of high level Petri nets, called reconfigurable nets, which dynamically modify their own structure. As far as dynamic change is concerned, the reconfigurable nets can be used to emulate synthetic cut over changes but fails in general to emulate jumpers. The reason is that the model does not allow the creation nor the disappearance of tokens, but only token movements. On the other hand, reconfigurable nets are better suitable than hybrid flow nets to support multiple *modes of operation.*

In [1], the authors are independently adopting a methodology similar to flow jumpers. Their dynamic correctness revolves around the notion of *safe state* w.r.t. a change. According to their model, a dynamic change occurs only if the state reached by a job (in the old procedure) is safe w.r.t. the change. To comply with this requirement, they propose *linear jumpers* as means to "force" a job into a safe state (in the old procedure.) There seems to be at least one fundamental difference in our respective approaches. Their model accommodates *retroactive changes*, in the sense that in some cases, getting to a safe state may require undoing some of the activities which have taken place. This gives rise to the not so trivial issue of the *undo semantics.*

In [25], the issue of workflow flexibility is addressed. The authors introduce *ad-hoc workflows* based on *process templates.* These process templates are considered as *reference models.* They also give a set of static structural transformations which may be used to build safe and successfully terminating workflow nets starting from a library of basic process templates which enjoy these properties.

In [24], the authors define process equivalence based on *delay bisimilarity.* Similarity between cases (i.e. jobs) is *conserved* by considering them as *extensions* or *reductions* of the same ancestor. In the event that a change results in a process extension, the change can be applied dynamically without delay to a running job. However, no mention is made if the change results in a process reduction.

5 Conclusions and Summary

Dynamic structural change to office procedures is a pervasive unsolved problem within workflow environments. This paper has introduced the timed flow nets as a way of accommodating time issues into the design of workflow systems and the analysis of their static changes. It has also expanded on the issue of "safe" static transformations which preserve the soundness properties.

This work has also briefly presented a new Petri-net based model, namely the timed hybrid flow nets, that is especially suitable to address workflow dynamic changes . In a companion technical report to this paper, we formally define timed hybrid flow nets, their semantics and their application to the problems of dynamic change. We also expand upon the results from [9], state and establish results concerning the dynamic change composition and iteration.

The issue of dynamic change correctness is currently under investigation in a broader context than in [9, 20]. This effort is concerned with the design and implementation of SL-DEWS, a specification language for the dynamic evolution of workflow systems. We hope that by the time of the 1998 Petri net conference in Portugal, we will have interesting results to report on SL-DEWS.

Acknowledgements We wish to thank the anonymous referees for their helpful comments which helped to improve the quality of this work and for pointing out some related references.

6 Mathematical Notations

In the course of this work, we make use of the following mathematical notations:

• $\mathcal{AN}$ denotes a finite alphabet of *activity names.* $\lambda \in \mathcal{AN}$, will be used to denote the *silent action.* $\mathcal{JN}$ denotes a finite alphabet of *job names.*

• $\mathbb{Q}^+$ denotes the set of positive rational numbers (including 0) to which we adjoin the infinity element, denoted ∞. Unless explicitly mentioned otherwise, $\mathbb{Q}^+$ will be used as the time domain of our model. $\mathbb{T}$ denotes the time interval domain each element of which is an interval over $\mathbb{Q}^+$ of the form $[a, b]$ (i.e. a closed interval) or $[a, \infty[$ (i.e. a semi-open interval) where $a, b \in \mathbb{Q}^+$. We will use $[a]$ as a shorthand notation for the *singleton* $[a, a]$. The addition over positive rational numbers is extended by taking the convention that $a + \infty = \infty + a = \infty$ and $\infty + \infty = \infty$. The order relation on positive rational numbers is extended as well by taking ∞ as the maximal element of $\mathbb{Q}^+$. For x and y in $\mathbb{T}$, x and y are *overlapping* iff $x \bigcap y$ is not a singleton. $x \otimes y$ is the time interval whose lower bound is the maximum of the lower bounds of x and y, and whose upper bound is the maximum of the uppers bound of x and y. $x \oplus y$ is the time interval obtained by taking the sum of the bounds.

• For a finite set A, $(A)_{MS}$ denotes the class of multi-sets over A, equipped with usual operators $\bigcup$, $\bigcap$ and $-$.

• For a finite alphabet A, A^* denotes the language of all strings (or equivalently ordered sequences) over A. λ denotes the empty word. For $w, w' \in A^*$, ww' denote the concatenation of w and w', $\underline{lgth}(w)$ denotes the length of w, (with $\underline{lgth}(\lambda) = 0$.) If $w \neq \lambda$ and $1 \leq i \leq \underline{lgth}(w)$, $w[i]$ denotes the ith element of w. A sequence u is a suffix of a sequence w if $w = uv$ for some sequence v. $\underline{Suffix}(w)$ denotes the set of suffixes of w. This notion is extended to a language. Thus, for $L \in A^*$, $\underline{Suffix}(L)$ denotes the language of all suffixes of all elements of L.

• Let A, B , C and D be finite sets. Let $f : A \longrightarrow B$ be a function (not necessarily total) from A to B. $\underline{domain}(f)$ denotes the domain of f and $\underline{range}(f)$ denotes the range of f. For $A' \subseteq A$, $f{\downarrow}_{A'}$ denotes the restriction of f over A'. f is said to be injective iff no two (distinct) elements of A map to the same element of B, in this case f^{-1} denotes the inverse function of f. f is said to be surjective iff $\underline{range}(f) = B$. Id_A denotes the identity function over A. For $g : B \longrightarrow C$, $g \circ f$ denotes the composition of g and f. If C and B are disjoint and $h : C \longrightarrow D$, then $f \bigcup h$ denotes the (disjoint) union of f and g.

• For $u = (x, y) \in A \times B$, x is denoted $\pi_1(u)$ and y is denoted $\pi_2(u)$. This notation is extended to tuples of arbitrary length using $\pi_1 \ldots \pi_n$ (as appropriate.)

References

1. A. Agostini, F. De Michelis. “Simple workflow models” In Proceedings of WFM98: Workflow Management: Net-Based Concepts, Models, Techniques and Tools, PN98, Lisbon, Portugal.
2. M. Ajmone Marsna, G. Balbo, A. Bobbio, C. Chiola, G. Conte, A. Cumani. “On Petri Nets with Stochastic Timing” In Proc. of the International Workshop on Timed Petri Nets, Torino, 1985, IEEE Computer Society Press.
3. M. Ajmone Marsna, G. Balbo, G. Conte. “A Class of Generalized Stochastic Petri Petri Nets for the Performance Evaluation of Multiprocessor Systems” ACM Transactions on Computer Systems, 2 (1984).
4. E. Badouel, J. Oliver. “Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynmaic Changes” In Proceedings of WFM98: Workflow Management: Net-Based Concepts, Models, Techniques and Tools, PN98, Lisbon, Portugal.
5. G. Berthelot, H. Boucheneb. “ Occurence Graphs For Interval Timed Coloured Nets” Application and Theory of Petri Nets 1994, Lecture Notes in Computer Science, volume 815, Springer-Verlag, 1994.
6. B. Berthomieu, M. Diaz. “ Modeling and verification of time depenedent systems using time Petri nets” IEEE Transactions on Software Engineering, vol. 17, No 3, March 1991.
7. G. De Michelis, and Ellis, C.A. Computer Supported Cooperative Work and Petri Nets. Third Advanced Course on Petri Nets, Dagstuhl Castle, Germany (1996). Springer Verlag Lecture Notes in Computer Science.
8. C. A. Ellis and Nutt, G.J. "Modeling and Enactment of Workflow Systems". In M. Ajmone Marsan, editor, Application and Theory of Petri Nets 1993, volume 691 of Lecture Notes in Computer Science, pages 1-16. Springer-Verlag, Berlin, 1993.
9. C.A. Ellis, Keddara, K and Rozenberg, G. "Dynamic Change within Workflow Systems". Proceedings of the Conference on organizational Computing systems, ACM Press, New York (1995) 10-21.
10. C. Guezzi, D. Mandrioli, S. Morasca, P. Mauro. “A general way to put time into Petri nets” In Proc. of the Fifth International Workshop on Software Specification, Vol. 14-3 of ACM SIGSOFT Engineering Notes, Pittsburg, Pennsylvania, USA, 1989.
11. C. Guezzi, S. Morasca, M. Pezze. “Validating Timing Requirements for TB Net Specifications” The Journal of Systems and Software, vo. 27, No 7, November 1994.
12. K. Jensen. “Coloured Petri Nets: Basic concepts, Analysis Methods and Practical use. volume 1: Basic Concepts”. EATCS Monographs on Theoretical Computer Science, Springer-Verlag 1992.
13. K. Keddara. “On the Dynamic Evolution of Workflow Systems” Ph.D. Thesis in preparation.
14. P. Merlin, D.J. Farber. “Recoverability of communication protocols”. IEEE Transactions on Communications, 24, 1976.
15. T. Murata. “Petri nets: properties, analysis, and applications. Proceedings of the IEEE 77(4), 1989.
16. C. Ramchandani “Analysis of Asynchroneous Concurrent Systems by Timed Petri Nets” Project MAC, TR 120, MIT, 1974
17. W. Reisig. “Petri Nets”. Springger 1985.
18. J. Sifakis. “Use of Petri Nets for Performance Evaluation”. Measuring, Modeling and Evaluating Computer Systems, H. Beilnerand E. Gelenbe editors, North Holland, 1977
19. H. Saastamoinen "On the Handling of Exceptions in Information Systems” University of Jyvaskyla PhD Dissertation, Nov. 1995.
20. W.M.P. van der Aalst. "Verification of Workflow Nets". In P. Azema and G. Balbo, editors, Application and Theory of Petri Nets 1997,volume 1248 of Lecture Notes in Computer Science, pages 407-426. Springer-Verlag, Berlin, 1997.
21. W.M.P. van der Aalst. "Finding Erros in the Design of a Workflow Process". In Proceedings of WFM98: Workflow Management: Net-Based Concepts, Models, Techniques and Tools, PN98, Lisbon, Portugal.
22. W.M.P. van der Aalst." Interval Timed Colored Petri Nets and their Analysis". Application and Theory of Petri Nets 1993, 14th International Conference, Chicago, Illinois, USA, LNCS 691, Springer-Verlag.
23. R. Vlak. "Self-Modifying Nets, a Natural Extension of Petri Nets". Proceedings of Icalp'78, Lecture Notes in Computer Science vol.62 (1978) 464-476.
24. M. Voorhoeve, W.M.P. Van der Aalst. “Conservative Adaption of Workflow” In M. Wolf and U. Reimer, editors, Proceedings of the International Conference on Practical Aspects of Knowledge Management (PAKM'96), Workshop on Adaptive Workflow, Basel, Switzerland, 1996
25. M. Voorhoeve, W.M.P. Van der Aalst. “Ad-hoc Workflow: Problems and Solutions” In R. Wagner, editor, Proceedings of the 8th DEXA Conference on Database and Expert Systems Applications, Toulouse, France, 1997.

26. WFMC. Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011) Technical Report, Workflow Management Coalition, Brussels, 1996.
27. W. Zuberek. "Timed Petri nets and preliminary performance evaluation". In Proc. 7th Annual Symposium on Computer Architecture, La Baule, France, 1980.
28. P. Bichler, G.. Preuner, M. Schrefl. "Workflow Transparency". In Proc on the Intl'l Conf. on Advanced Information Systems (CAiSE 97).

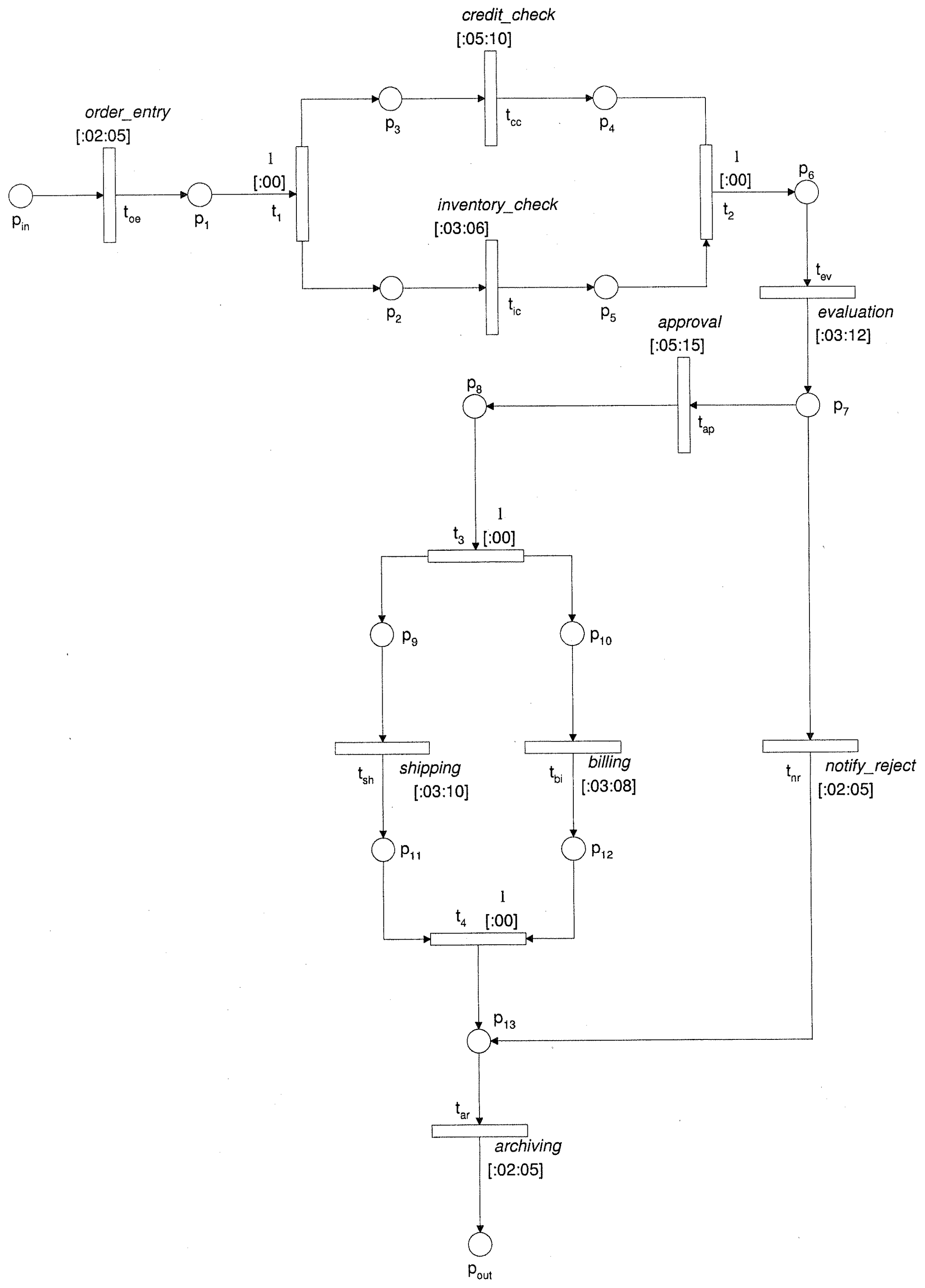


Fig. 1. A version $OrderProc_1$ of the order processing procedure.

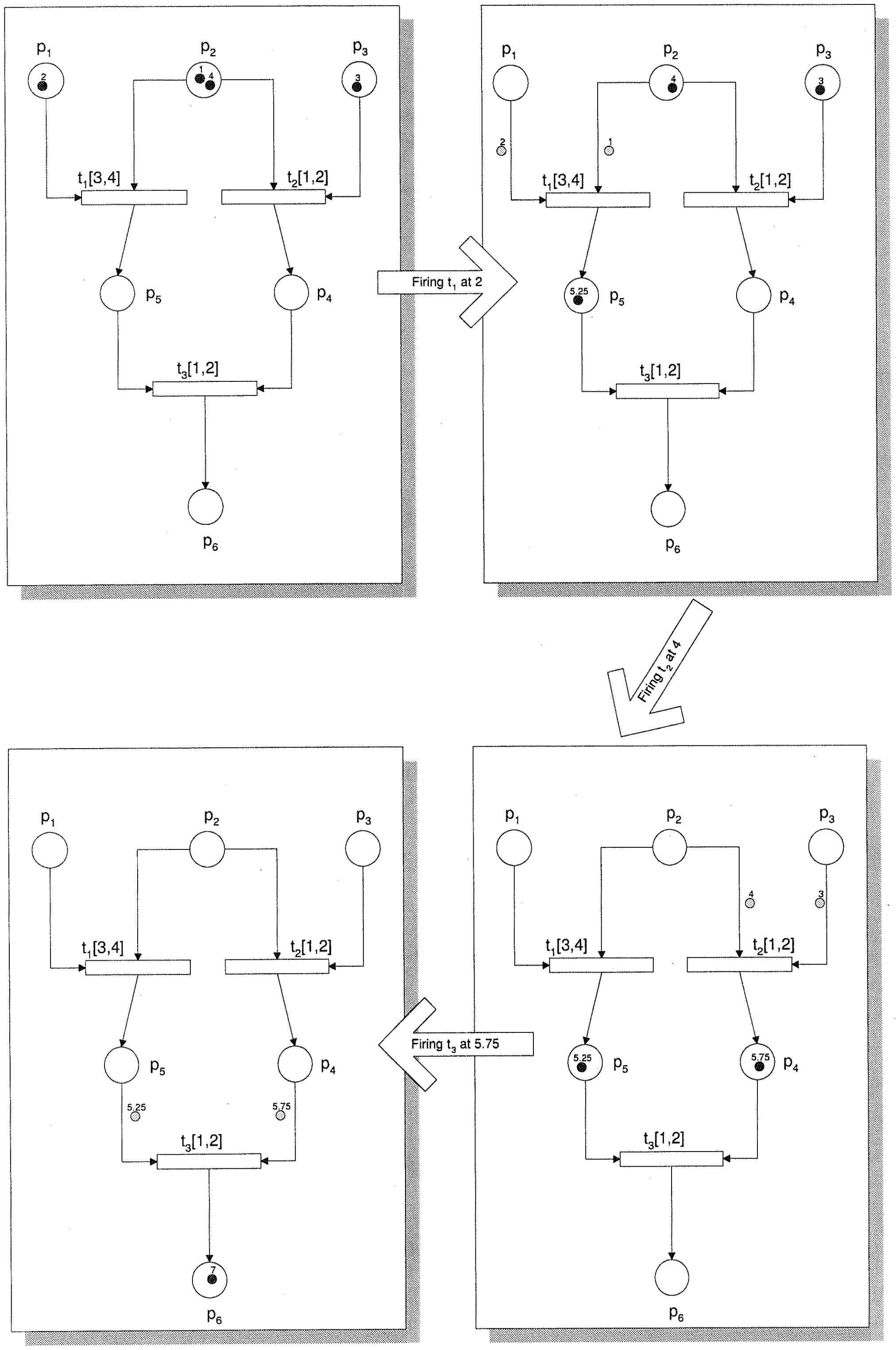


Fig. 2. Ilustration of the event firing semantics.

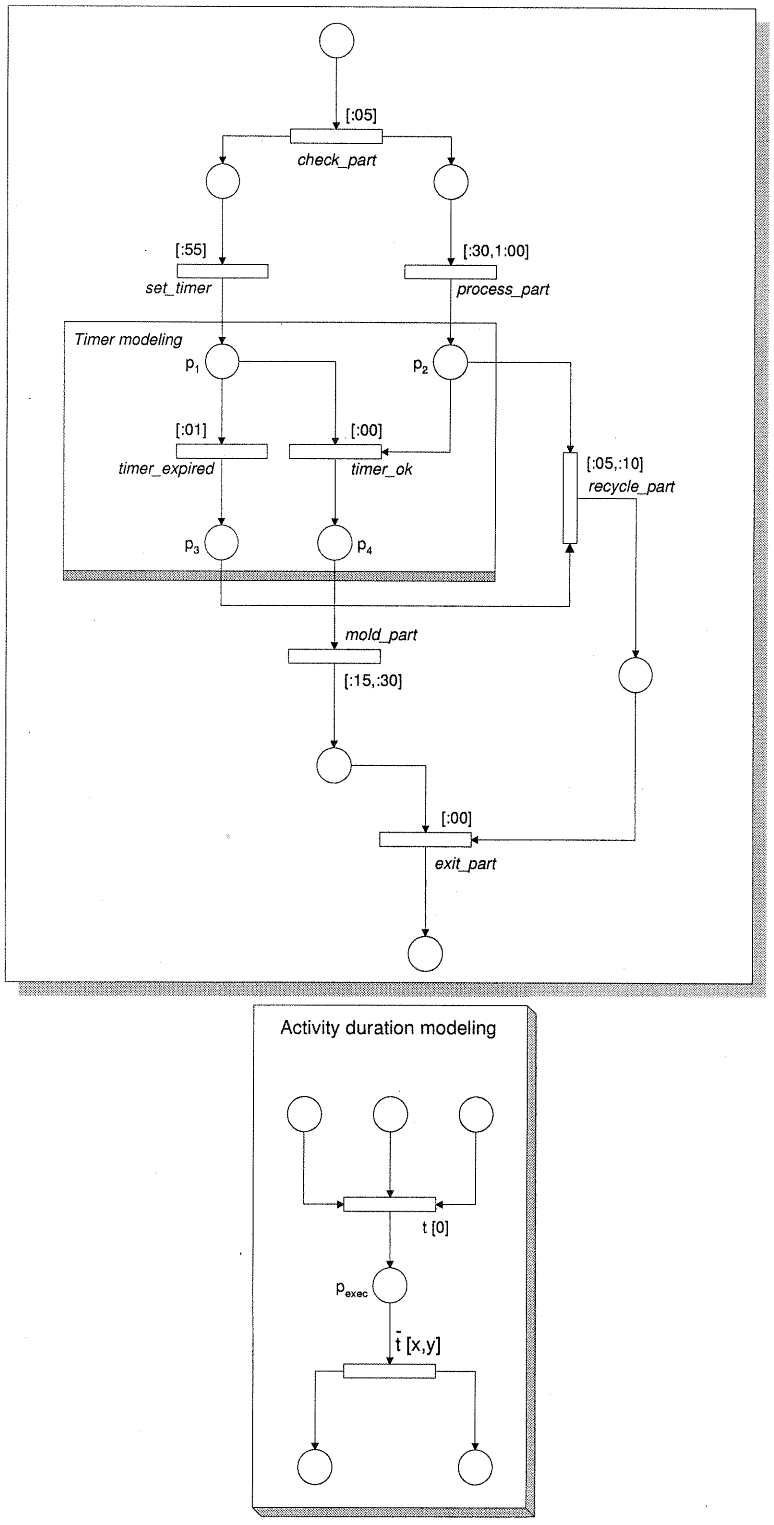
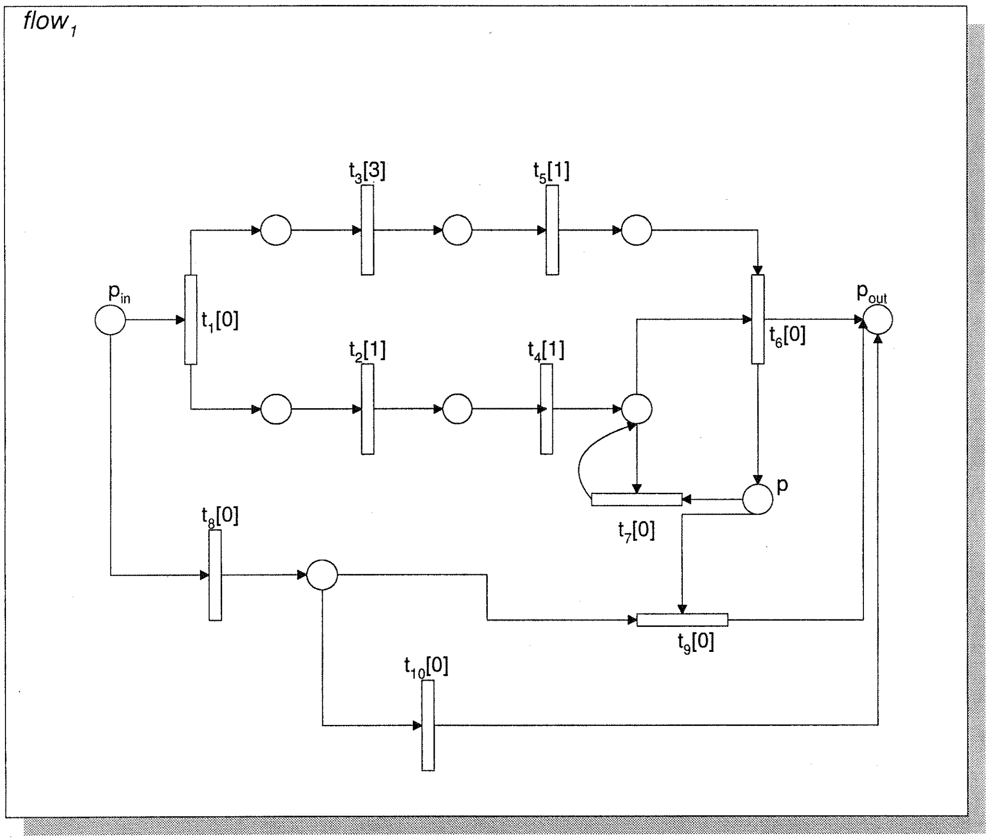


Fig. 3. Timer modeling and activity duration modeling.



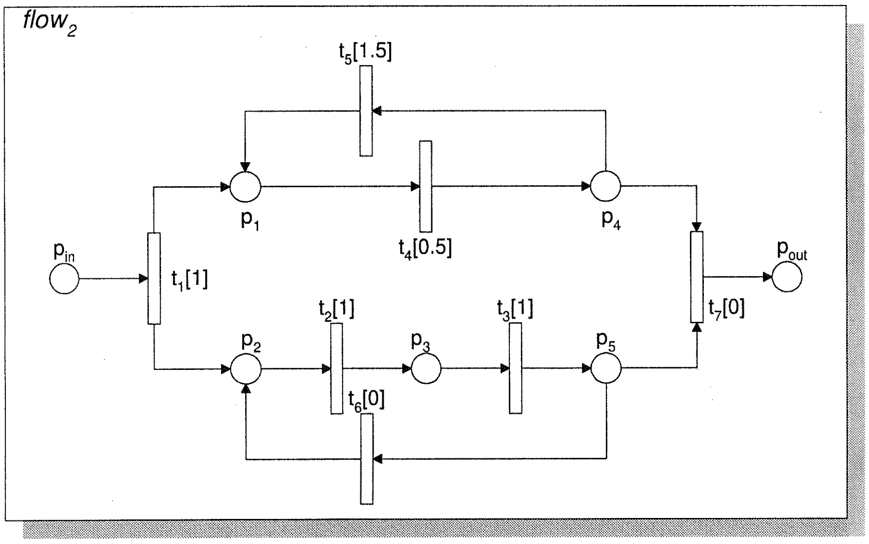


Fig. 4. *flow*$_1$: the converse of (8) does not hold. *flow*$_2$: soundness does not carry from the untimed structure of a timed flow net.

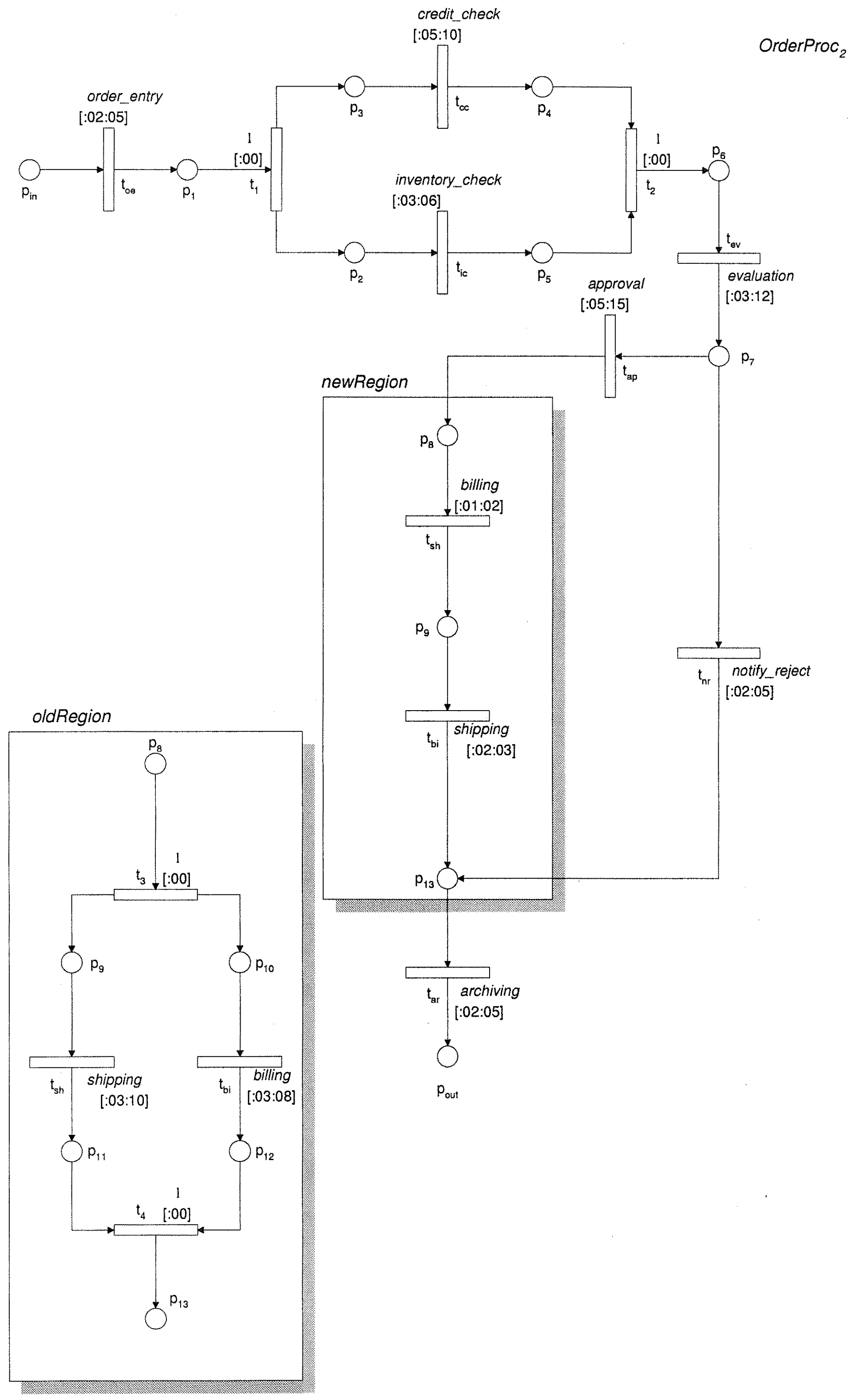


Fig. 5. *Change*$_1$: Sequentialization of *billing* and *shipping*; *OrderProc*$_2$ models the new version

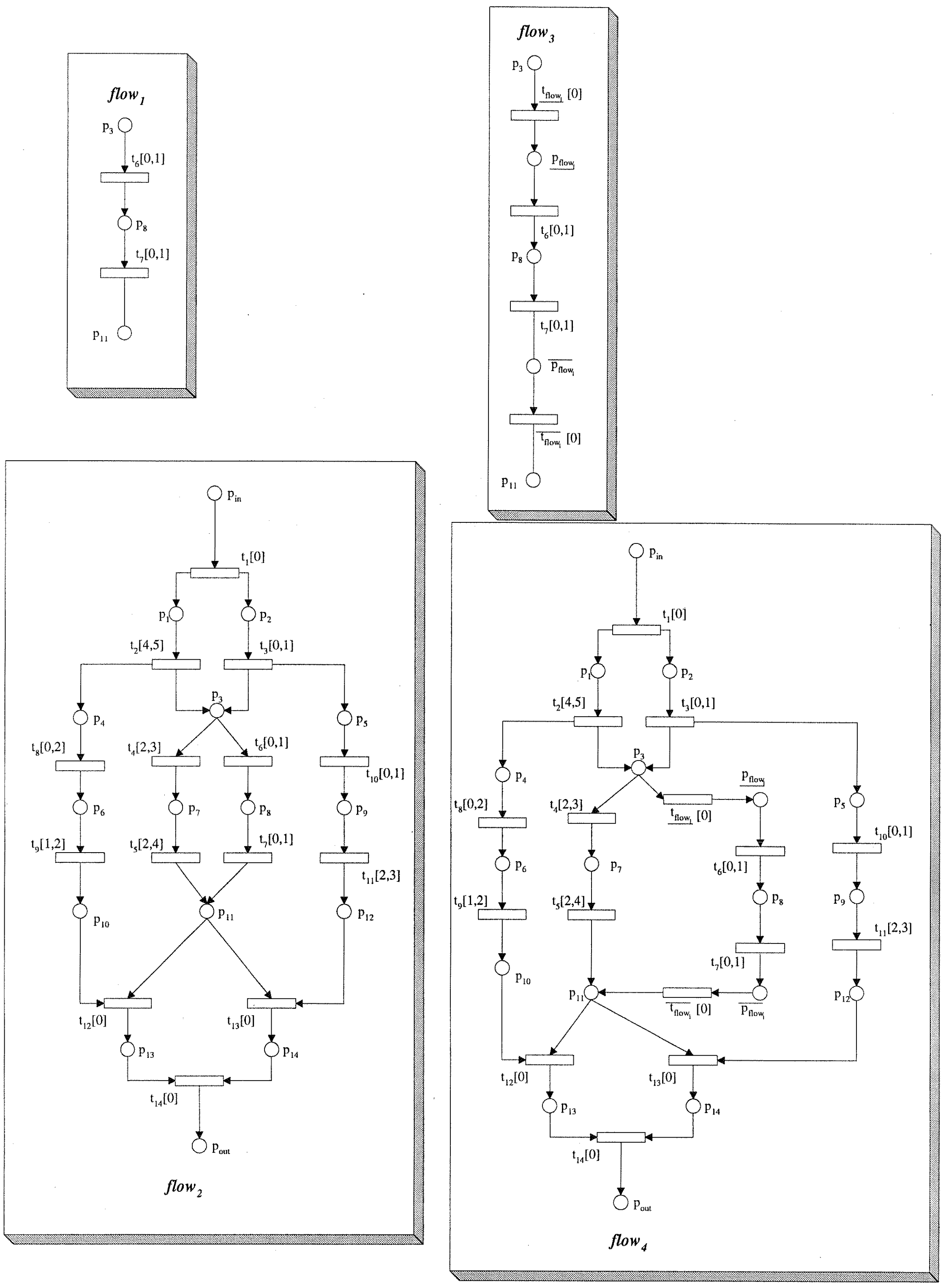
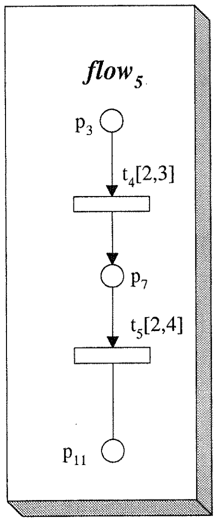
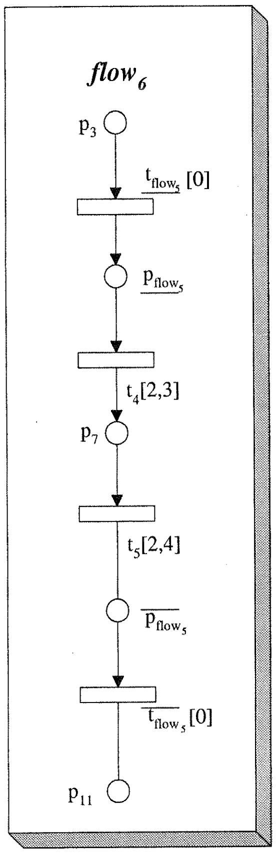


Fig. 6. The seriability property does hold





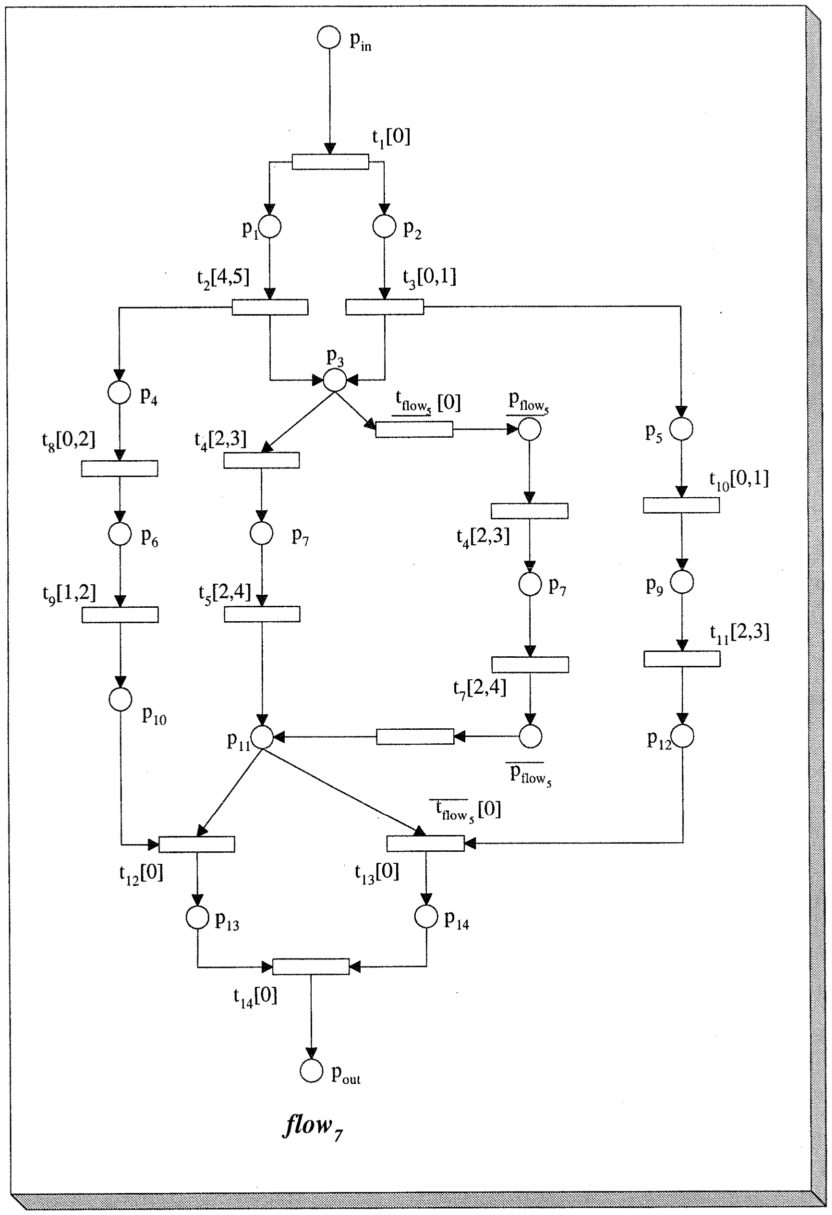


Fig. 7. The seriability property does not hold.

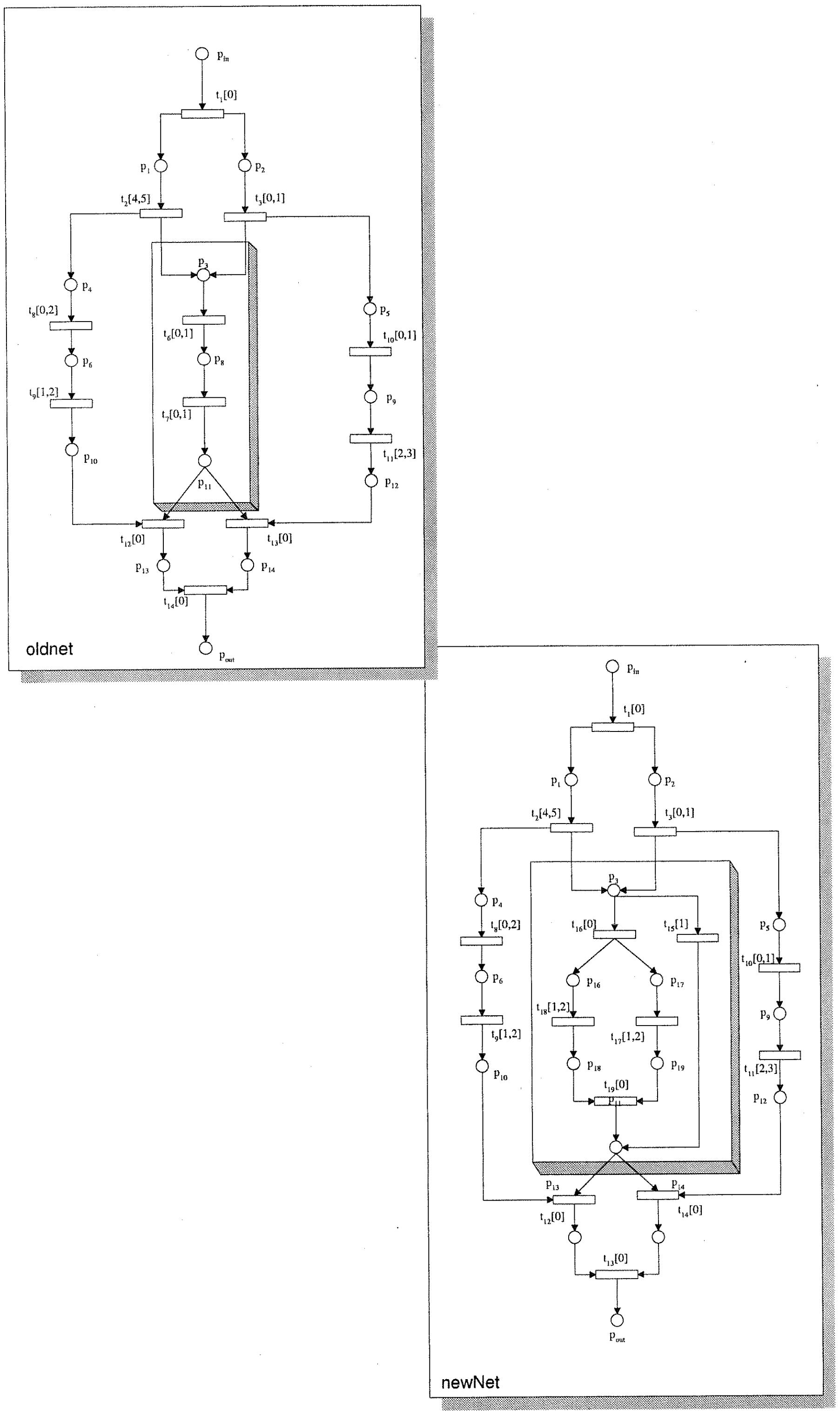


Fig. 8. Ilustration of the segment iteration.

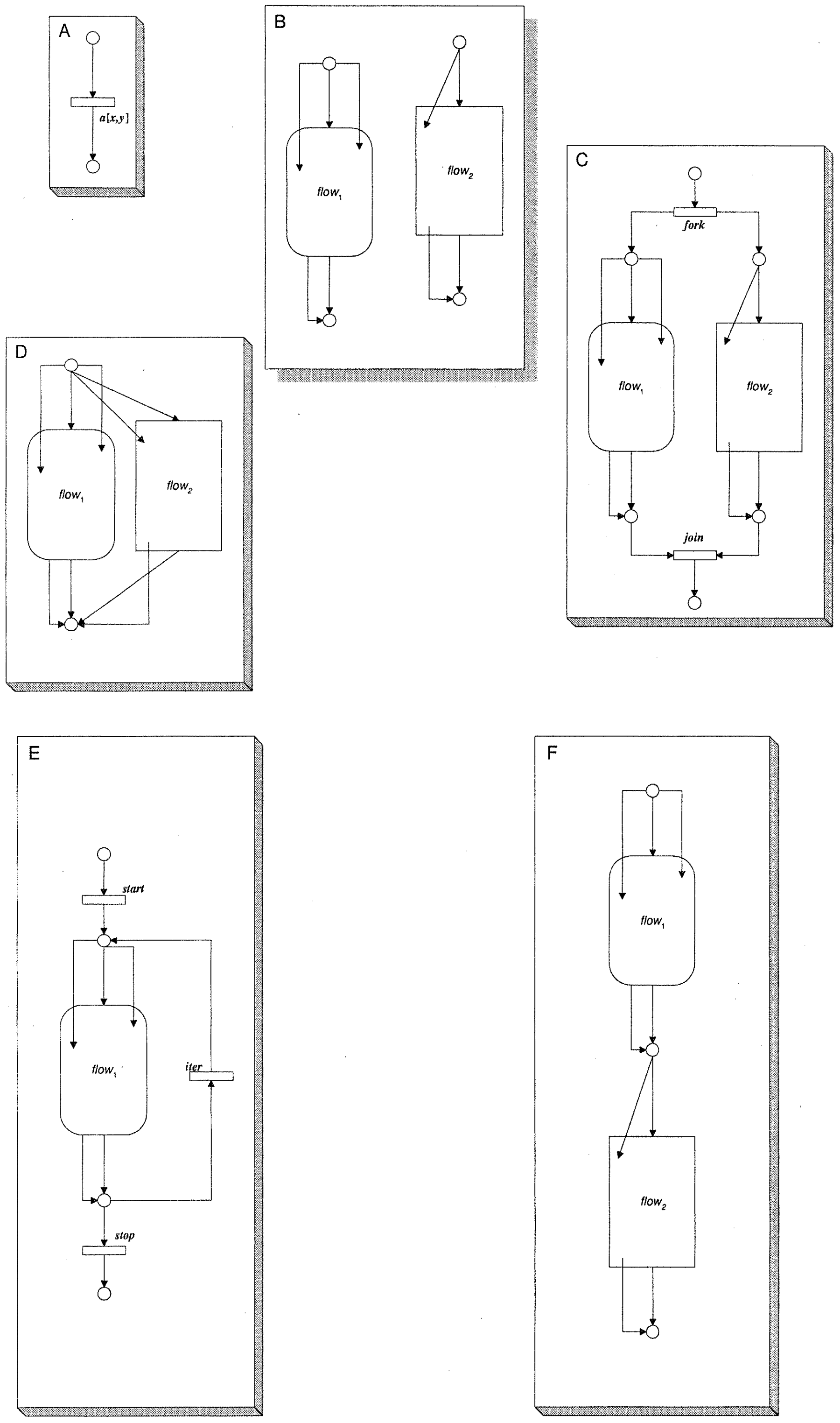
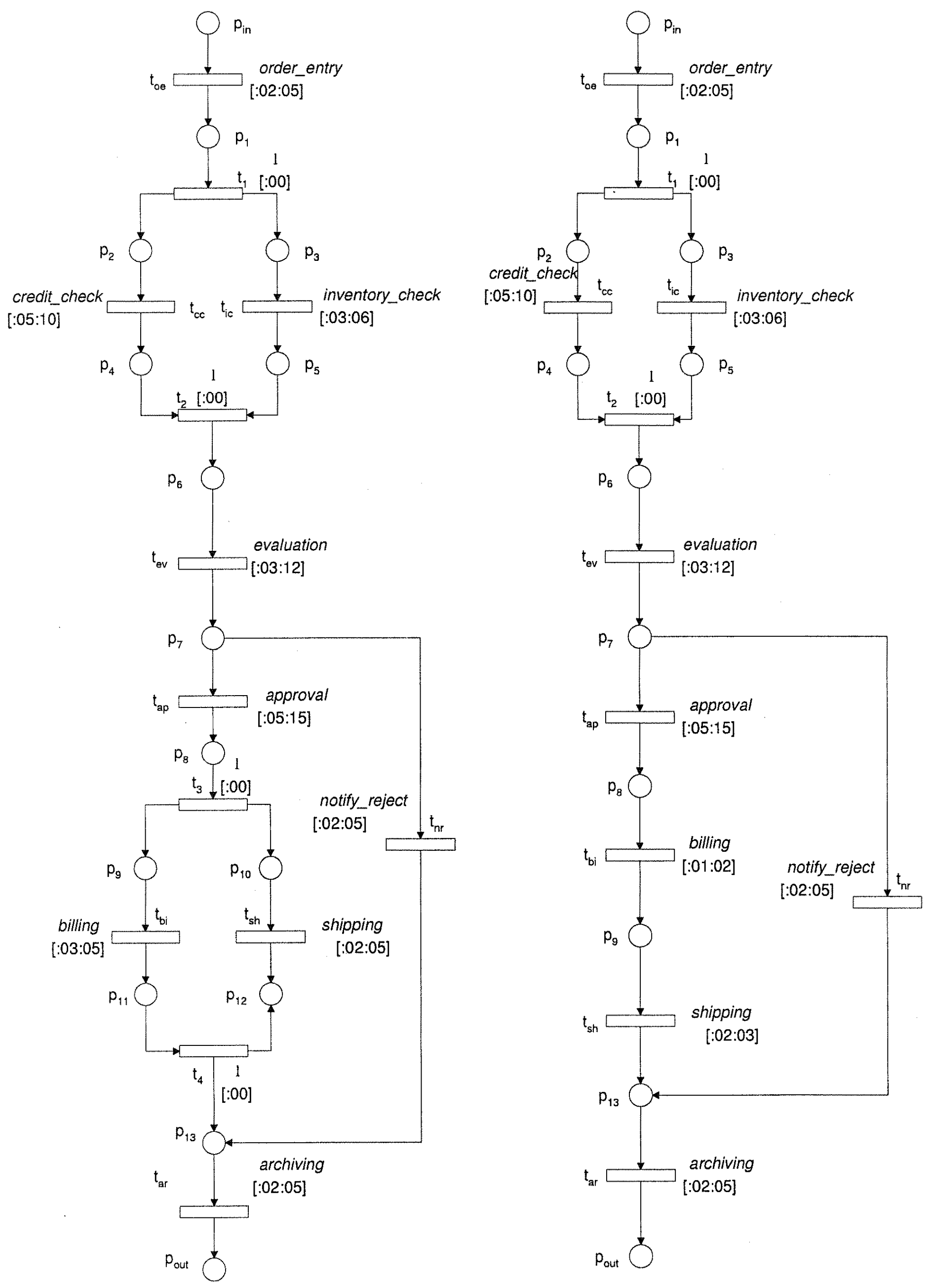


Fig. 9. Illustration of the sound flow transformations.



Jumper	*jmp*$_{in}$	*jmp*$_1$	*jmp*$_2$	*jmp*$_3$	*jmp*$_4$	*jmp*$_5$	*jmp*$_6$	*jmp*$_7$	*jmp*$_8$	*jmp*$_9$	*jmp*$_{10}$	*jmp*$_{11}$	*jmp*$_{out}$
Type	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*
Priority	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
Inlets	p_{in}	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9, p_{10}	p_{11}, p_{10}	p_{13}	p_{out}
Outlets	p_{in}	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_8	p_9	p_{13}	p_{out}

Fig. 10. Dynamic change modeling is based on flow jumpers.

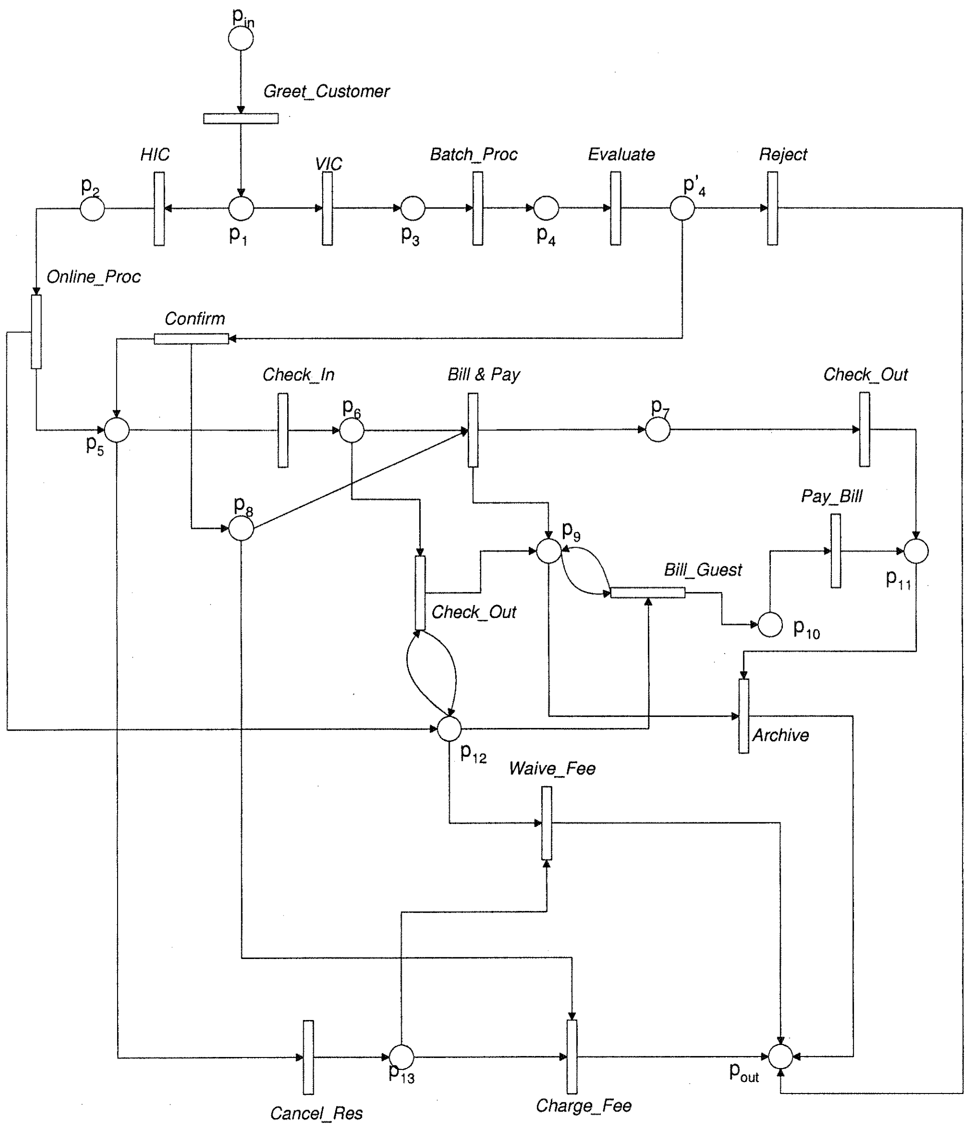


Fig. 11. *DIBp*$_1$: A first version of the *Desert Inn* workflow procedure

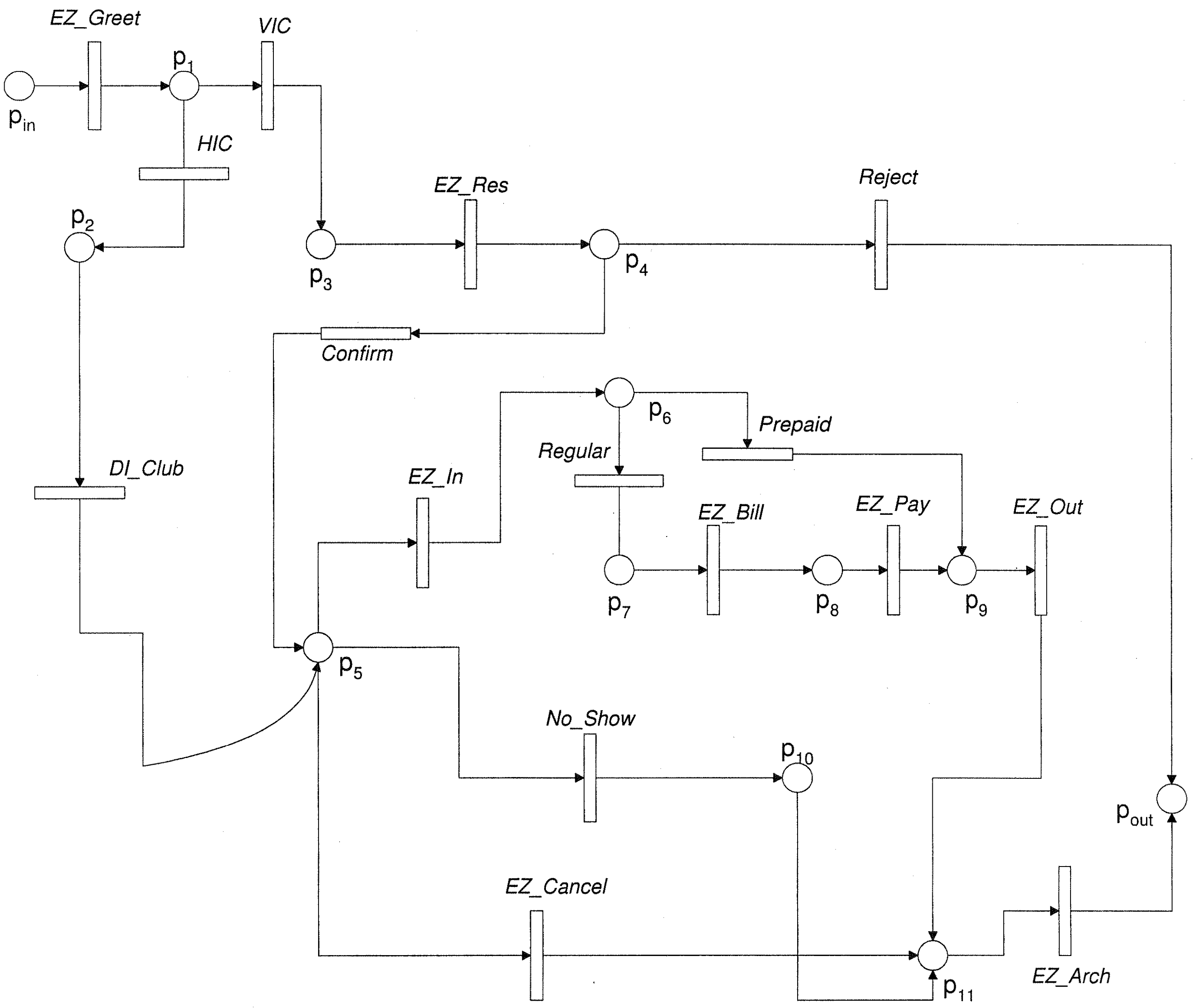
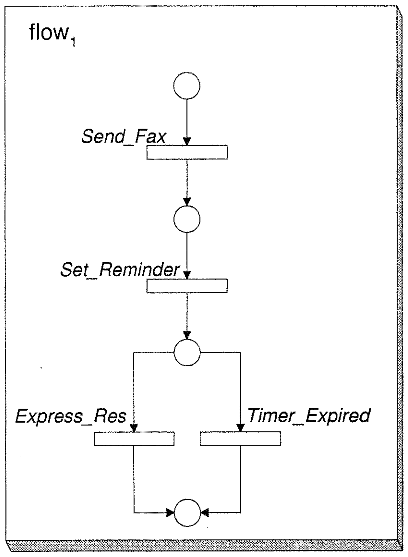
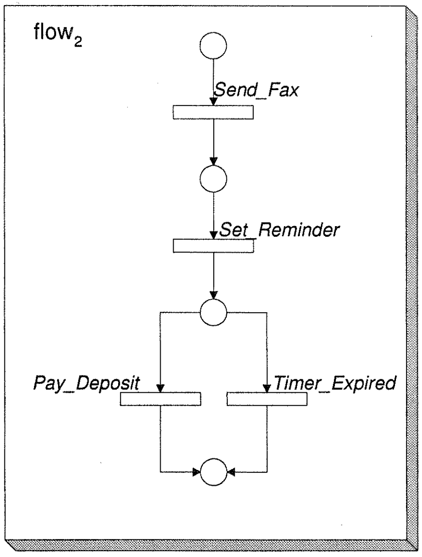
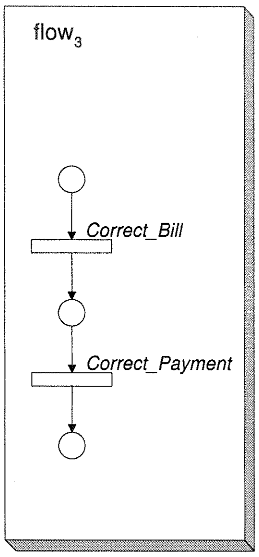
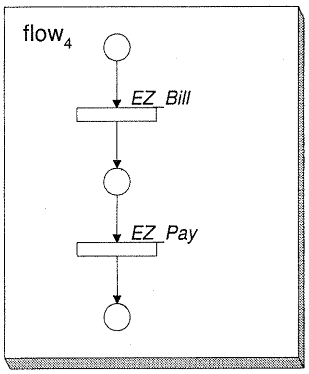


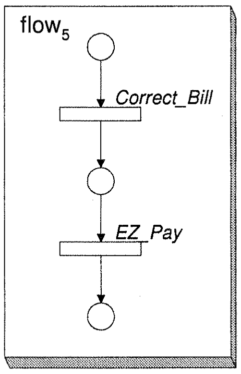
Fig. 12. *DIBp*$_2$: An enhanced version of the *Desert Inn* workflow procedure

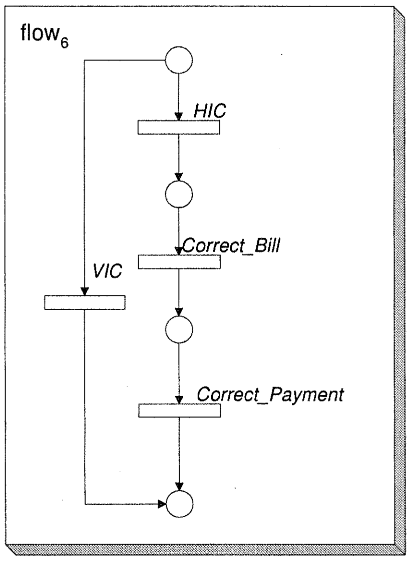












Macro	*macro$_1$*	*macro$_2$*	*macro$_3$*	*macro$_4$*	*macro$_5$*	*macro$_6$*
Bridge	*flow$_1$*	*flow$_2$*	*flow$_3$*	*flow$_4$*	*flow$_5$*	*flow$_6$*
Priority	*1*	*1*	*1*	*1*	*1*	*1*
Inlets	p_3	p_5, p_8	p_7, p_9	p_{12}, p_9	p_9, p_{10}	p_9, p_{11}
Outlets	p_4	p_4	p_9	p_{11}	p_{11}	p_{11}

Jumper	*jmp$_{in}$*	*jmp$_1$*	*jmp$_2$*	*jmp$_3$*	*jmp$_4$*	*jmp$_{out}$*
Type	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*	*mvr*
Priority	*1*	*1*	*1*	*1*	*1*	*1*
Inlets	p_{in}	p_5, p_{12}	p_6, p_8	p_8, p_{13}	p_{12}, p_{13}	p_{out}
Outlets	p_{in}	p_2	p_7	p_{11}	p_{11}	p_{out}

Fig. 13. Implementation of *The Desert Inn* migration policy