Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

Designing Dependable Dynamic Workflows through a Reflective PN-based Approach

Lorenzo Capra *

* *Department of Informatics and Communication, Università degli Studi di Milano, 20139 Milan, ITALY (Tel: +39-02-50316256; e-mail: capra@dico.unimi.it).*

Abstract: The management of dynamic workflows needs adequate formal models and support tools to handle in a safe way changes occurring during workflow operation. A common approach is to pollute models with details that do not regard the workflow behavior, rather its evolution. That hampers analysis, reuse and maintenance in general.
We propose and discuss the adoption of a recent Petri net-based reflective model to support dynamic workflow design, by addressing a localized open problem: how to determine what tasks should be redone and which ones do not when transferring a workflow instance from an old to a new template. The idea behind is that keeping functional aspects separated from evolutionary ones, applying evolution to a workflow template only when necessary, results in a clean reference model of dynamic workflows on which the ability of verifying major workflow properties favors a dependable evolution.

1. INTRODUCTION

Business processes are frequently subject to change due to two main reasons (van der Aalst and Jablonski [2000]): i) at design time the workflow specification is incomplete due to lack of knowledge, ii) errors or exceptional situations can occur during the workflow execution; these are usually tackled on by deviating from the static schema, and may cause breakdowns, reduced quality of services, and inconsistencies.

Most of existing Workflow Management Systems -WMS in the sequel (e.g., IBM Domino, iPlanet, Fujisu iFlow, TeamCenter) are designed to cope with static processes. The commonly adopted policy is that, once process changes occur, new workflow templates are defined and workflow instances are initiated accordingly from scratch. This oversimplified approach forces tasks that were completed on the old instance to be executed again, also when not necessary. If the workflow is complex and/or involves a lot of external collaborators, a substantial business cost will be incurred.

Dynamic workflow management might be brought in as a solution. Formal techniques and analysis tools can support the development of WMS able to handle undesired results introduced by dynamic change.

In the research on dynamic workflows, the prevalent opinion is that models should be based on a formal theory and be as simple as possible. In Agostini and De Michelis [2000] process templates are provided as 'resources for action' rather than strict blueprints of work practices. May be the most famous dynamic workflow formalization, the ADEPTflex system (Reichert and Dadam [1998]), is designed to support dynamic change at runtime, making at our disposal a complete and minimal set of change operations. The correctness properties defined by ADEPTflex are used to determine whether a specific change can be applied to a given workflow instance or not.

Most of workflow modeling techniques are based on Petri Nets (PN) (Salimifard and Wright [2001]), due to PN's description efficacy, formal essence, and the availability of consolidated PN-based analysis techniques. Classical PN have a fixed topology, so they are well suited to model workflow matching a static paradigm. Conversely, concerns relating to dynamism/evolution must be hard-wired in classical PN and bypassed when not in use. That requires some expertise in PN modeling, and might result in incorrect or partial descriptions of workflow behavior. Even worst, analysis would be polluted by a great deal of details concerning evolution.

Separating evolution from (current) system behavior is worthwhile. This concept has been recently applied to a PN-based context (Capra and Cazzola [2005]), using reflection (Maes [1987]) as mechanisms that easily permits separation of concerns. A basic reflective model layered in two causally connected levels (base-, and meta-) is used.

With respect to several dynamic PN extensions recently appeared (Cabac et al. [2005], Hoffmann et al. [2005], and as concerns specifically the workflow field Badouel and Oliver [1998], Ellis and Keddara [2000], Hicheur et al. [2006]) reflective Petri nets (Capra and Cazzola [2005]) are not a new PN class, rather they rely upon classical PN. That gives the possibility of using available tools and consolidated analysis techniques.

We propose reflective PN as formal model supporting the design of sound dynamic workflows. A structural characterization of sound dynamic workflows is adopted, based on PN's free-choiceness preservation. The approach is applied to a localized open problem: how to determine what tasks should be redone and which ones do not when transferring a workflow instance from an old to

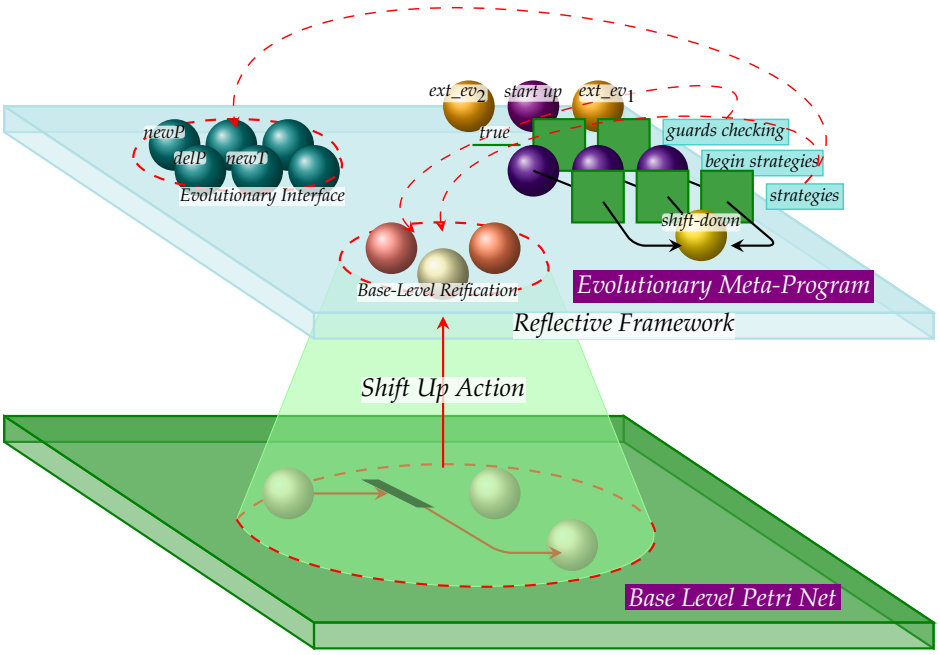


Fig. 1. Snapshot of the reflective model: the base-level prone to be adapted, the meta-level driving the adaptation.

a new template. The problem is efficiently but rather empirically addressed in Qiu and Wong [2007], according to a template-based schema relying on the concept of *bypassable* task. Conforming to the same concept we propose an alternative, that allows evolutionary steps to be soundly formalized, and basic workflow properties to be efficiently verified.

As it is widely agreed (Agostini and De Michelis [2000]), the workflow model is kept as simple as possible. Our approach has some resemblance also with Reichert and Dadam [1998], sharing the completeness/minimality criteria, even if it considerably differs in management of changes: it neither provides exception handling nor undoing mechanism of temporary changes, rather it relies upon a sort of "on-the-fly" validation.

The balance of the paper is as follows: in section 2 we outline the PN-based reflective model, and the adopted terminology; in section 3 we state some basic notions around PN and workflows; in section 4 we sketch the template-based dynamic workflow approach (Qiu and Wong [2007]) that we are comparing to; in section 5 we present our alternative based on reflective PN, using the same example as in Qiu and Wong [2007]; last we draw some conclusions and perspectives.

2. THE REFLECTIVE PN MODEL

The *reflective Petri net* model (Capra and Cazzola [2005]) permits designers to formalize a discrete-event system and *separately* its possible evolutions, and to dynamically adapt the system model when evolution is necessary.

The approach is based on a reflective architecture (Fig. 1) structured in two logical layers. The *base-level* is an *ordinary* Petri net (Reisig [1991]) modeling a system prone to evolve (in this context a workflow), called *base-level PN*; whereas the second layer is the *meta-level*, or *meta-program* following the reflection parlance, a *high-level* PN (Jensen and Rozenberg [1991]) composed by the *evolutionary strategies* that will drive the evolution of the base-level PN when certain events occur.

The *reflective framework*, realized by a high-level PN as well, is responsible for carrying out the evolution of the base-level PN at the meta-level. Meta-level computations in fact operate on a representative of the lower-level, called *reification*. The base-level PN reification is defined as a *marking* of the reflective framework, and is updated every time the base level Petri net enters a new state. The reification is used by the meta-program to observe (*introspection*) and manipulate (*intercession*) the base-level PN. After a meta-computation any change made to the reification is reflected to the base-level (*shift-down*), thanks to the one-to-one correspondence which is set between these entities.

According to the reflective paradigm, the base-level PN runs irrespective of the meta-program, being not aware of the existence of a meta-level. The meta-level is implicitly activated (*shift-up*), then a suitable strategy is put into action, i) either when the base-level PN model reaches a given configuration, or ii) when triggered by an external/unpredictable event. Each strategy locally acts on a precomputed influence area, a base-level region which is temporarily frozen for the sake of consistency while the strategy is being executed.

Intercession on the base-level PN is carried out in terms of a minimal but complete set of low-level transformations (the *evolutionary interface*), taking effect both on base-level's structure and current state (marking). The

evolutionary strategy specifies arbitrary transformation patterns. To make it easier the strategy design, meta-programmers have been provided with a simple meta-language whose syntax is inspired by Hoare's CSP (Hoare [1985]), enriched with a few ad-hoc constructs for manipulation of nets. A strategy can be automatically translated into a high-level PN, that in turn interacts to the evolutionary framework.

Evolutionary strategies are *transactions*: either they succeed, or leave the base-level unchanged. Several strategies could be candidate for execution at a given instant: different policies, ranging from full non-determinism to static priorities, could be adopted to select one.

The whole reflective PN structure, as well as the interaction between base- and meta-level and between meta-level entities, have been fully defined in (Capra and Cazzola [2005]). Let us outline the essential aspects:

— the reflective framework structure is fixed, while the evolutionary strategies are coupled to the base-level, so they vary from time to time;
— the reflective framework and the meta-program share the base-level reification and the evolutionary interface;
— the meta-program manipulates the base-level PN reification; the evolutionary interface records the transformation commands issued by the evolutionary strategies to the reflective framework (that operates them);
— the base-level reification data sets are like formal parameters, that are bound from time to time to a given base-level PN; reification's initial marking corresponds to the initial base-level configuration.

The fixed part of the reflective architecture is responsible for the reflective behavior, hiding the work of the evolutionary component to the base-level PN: that permits a clean separation between the models of evolution and evolving system, and avoids the base-level PN being polluted by details related to evolution.

3. WORKFLOW PETRI NETS

This section introduces terminology and notations for the base-level Petri net class used in the sequel. Basic concepts and properties related to workflow modeling are also given. We refer to Reisig [1991], van der Aalst [1996] for more elaborate introductions.

Definition 1. (Petri net) A Petri net is a triple $(P;T;F)$:

- P is a finite set of places,
- T is a finite set of transitions ($P \cap T = \emptyset$;),
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation)

Symbols ${}^\bullet n, n^\bullet$ denote the pre/post sets of $n \in P \cup T$, respectively. The extensions ${}^\bullet A, A^\bullet$, $A \subseteq P \cup T$ will be also used. In the workflow context it makes no sense to have weighted arcs, because places correspond to conditions. A marking (state) M is a distribution of tokens over places, $M \in Bag(P)$.

Transitions change the state of the net according to the following firing rule:

-A transition t is said to be enabled in M if and only if each place $p \in {}^\bullet t$ contains at least one token.

-An enabled transition t *may* fire, consuming one token from each $p \in {}^\bullet t$ and producing one token for each $p \in t^\bullet$.

Given $PN = (P;T;F)$ and marking M_i, we have the following notations:

$M_1 \xrightarrow{\sigma} M_n$: the firing sequence $\sigma = t_1 t_2 t_3 \dots t_{n-1}$ leads from M_1 to M_n via $M_2, \dots, M_{n-1}$.

M_n is *reachable* from M_1 if and only if $\exists \sigma$, $M_1 \xrightarrow{\sigma} M_n$.

$(PN; M_0)$ denotes a Petri net with an initial state M_0. Given $(PN; M_0)$, M' is said reachable if and only if it is reachable from M_0.

Let us define a few standard properties for Petri nets. First, we define properties related to the dynamics of a Petri net, then we give some structural properties.

(Live). $(PN; M_0)$ is live if and only if, for every reachable state M' and every transition t there exists M'' reachable from M' which enables t.

(Bounded, safe). $(PN; M_0)$ is bounded if and only if for each place p there exists $n \in \mathbb{N}$ such that for every reachable state M, $M(p) \leq n$. A bounded net is safe if and only if $n = 1$. A marking of a safe PN is denoted by a set of places.

(Path). A path C from a node n_1 to a node n_k of PN is a sequence $n_1, n_2, \dots, n_k$ such that $(n_i, n_{i+1}) \in F$, $1 \leq i \leq k-1$.

(Conflict). t_1 and t_2 are in conflict if and only if

${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$.

(Free-choice). PN is free-choice if and only if, for every pair of transitions t_1 and t_2, ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset \Rightarrow {}^\bullet t_1 = {}^\bullet t_2$.

(Causal connection - CC). transition t_1 is causally connected to t_2 if and only if $(t_1{}^\bullet \setminus {}^\bullet t_1) \cap {}^\bullet t_2 \neq \emptyset$.

3.1 Sound Workflow-nets and Free-Choiceness

A Petri net can be used to specify the control flow of a workflow. Tasks are modeled by transitions and causal dependencies by places and arcs. A place corresponds to a task pre/post-condition.

Definition 2. (Workflow-net). A Petri net PN $= (P;T;F)$ is a Workflow-net (hereafter WF-net) if and only if:

❶ There is one source place i such that ${}^\bullet i = \emptyset$.
❷ There is one sink place o such that $o^\bullet = \emptyset$.
❸ Every node $x \in P \cup T$ is on a path from i to o.

A WF-net has exactly one input place (i) and one output place (o), because a WF-net specifies the life-cycle of a case. The third requirement in definition 2 avoids dangling tasks and/or conditions, i.e., tasks and conditions which do not contribute to the processing of cases.

If we add to a WF-net PN a transition t^* such that ${}^\bullet t^* = o$ and $t^{*\bullet} = i$, then the resulting Petri net $\overline{PN}$ (called the *short-circuited* net of PN) is strongly connected.

The requirements stated in definition 2 only relate to the structure of the Petri net. However, there is another requirement that should be satisfied:

Definition 3. (soundness) A procedure modeled by a WF-net PN = $(P;T;F)$ is sound if and only if:

❶ For every state M reachable from state $\{i\}$, there exists σ, $M \xrightarrow{\sigma} \{o\}$
❷ $\{o\}$ is the only state reachable from $\{i\}$ with at least one token in place o.
❸ There are no dead transitions in $(PN;\{i\})$: $\forall t \in T$, there exists M reachable from $\{i\}$, $M \xrightarrow{t} M'$

In other words, for any case the procedure will terminate eventually (in the context of workflow we reasonably assume a strong notion of fairness) and the moment the procedure terminates there is a token in place o and all the other places are empty. That is sometimes referred to as *proper termination*. Moreover, it should be possible to execute any tasks by following the appropriate route though the WF-net.

The soundness property relates to the dynamics of a WF-net, and may be considered as a basic requirement for a process. A WF-net PN is sound if and only if $(\overline{PN};\{i\})$ is *live and bounded* (van der Aalst [1996]). Despite that useful characterization, it may be intractable to decide soundness for arbitrary WF-nets: liveness and boundedness are decidable, but also EXPSPACE-hard.

Therefore some structural characterizations of sound WF-nets were investigated (van der Aalst [1996]). Free-Choice (FC) Petri nets seem a good compromise between expressive power and 'analyzability'. They are the largest class of Petri nets for which strong theoretical results and efficient analysis techniques exist (Desel and Esparza [1995]). In particular (van der Aalst [1996]), soundness of a FC WF-net (as well as many other problems) can be decided in *polynomial* time. Moreover, a sound FC WF-net $(PN;\{i\})$ is guaranteed to be *safe*, according to the interpretation of places as conditions.

Another good reason for restricting to FC WF-nets is that the routing of a case should be independent of the order in which tasks are executed. If non-FC Petri nets were admitted, then the solution of conflicts could be influenced by the order in which tasks are executed. In literature the term confusion is often used to refer to a situation where the FC property is violated by a badly mixture of parallelism and choice.

Free-choiceness is a desirable property for workflow. If a process can be modeled as FC WF-net, one should do so. Most of existing WMS support FC processes only. We will admit as base-level Petri nets FC WF-nets.

Although FC WF-nets are a satisfactory characterization of well-structured workflows, there are non-FC WF-nets which correspond to sensible processes. S-coverability (van der Aalst [1996]) is a generalization of FC property: a sound FC WF-net is S-coverable. Unfortunately, it is impossible to verify soundness of an arbitrary S-coverable WF-net in polynomial time, this problem being PSPACE-complete.

4. A TEMPLATE-BASED DYNAMIC APPROACH

An interesting solution to facilitate efficient dynamic workflow change is proposed in Qiu and Wong [2007]. WMS supporting dynamic workflow change can either directly modify the affected instance, or restart it on a new workflow template. The first method is instance based while the second is template based. The approach we are describing, according to a consolidated practice, falls in the second category, and is implemented in Dassault Systèmes's SmarTeam, a PLM (Product Lifecycle Management) system including a WMS module. The idea consists of identifying all *bypassable* nodes, i.e., all nodes in the new workflow instance that satisfy the following conditions: i) they are unchanged, ii) they have finished in the old workflow instance, and iii) they need not be re-executed.

Two nodes (transitions in PN) are identical, before and after change, if and only if they represent the same task and preserve input/output connections. We hereafter assume that two nodes represent the same task if and only if they preserve name. To determine if a node/task is bypassable when the instance is transferred to a new template, an additional constraint is needed: all nodes from which there is a path to the node itself, must be bypassable themselves. A smart algorithm recognizes bypassable nodes: starting from the start node, bypassable by default, only successors of bypassable ones are considered.

This solution has been implemented in SmarTeam system, that includes a workflow manager and a messaging subsystem, but no built-in mechanisms facing dynamic workflow change. A set of API enables detaching and attaching operations between processes and workflow templates. A process is re-executed entirely if its template is changed. To realize workflow change, a server-application executes the following steps:

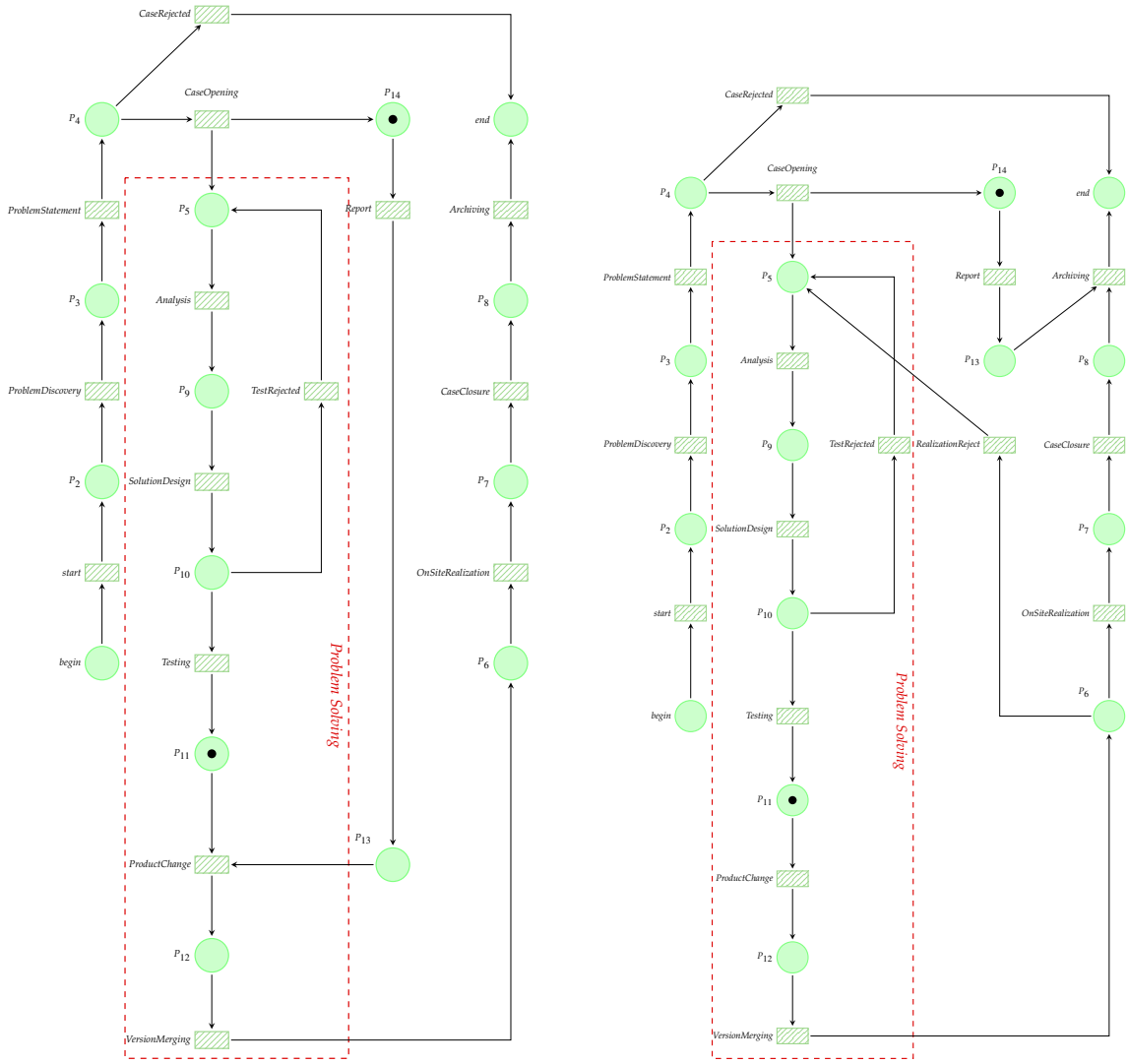
❶ obtain a process instance;
❷ obtain the old and new workflow templates;
❸ attach the new workflow template to the process;
❹ identify and mark the nodes that can be bypassed in the new workflow instance;
❺ initiate the new workflow without re-executing the marked nodes.

What appears unspecified in Qiu and Wong [2007] is how to safely operate steps ❹ and ❺: some heuristics are implemented, rather than a well defined methodology. No formal test is carried out to check the soundness of a workflow instance running on the modified template.

5. AN ALTERNATIVE BASED ON REFLECTIVE PN

Our alternative to Qiu and Wong [2007] is based on reflective PN. It allows a formalization of evolutionary steps, as well as a validation by means of PN structural analysis of changes proposed for the workflow templates. Validation is accomplished "on-the-fly", i.e., on the workflow reification while change is in progress. In particular, free-choiceness preservation is guaranteed. Changes are not reflected to the base-level in case of a negative check.

We consider the same application case presented in Qiu and Wong [2007]. A company has several regional branches. To enhance operation consistence, the company head-



(a) The base workflow template (b) Template's evolution

Fig. 2. A basic template and its possible evolution.

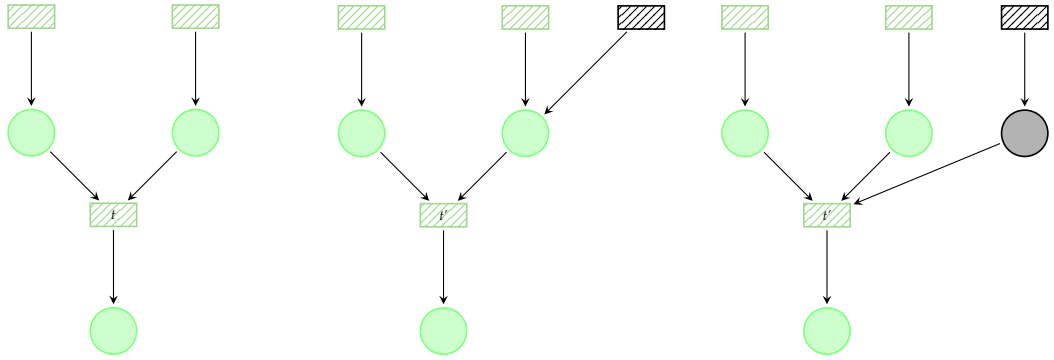
quarter (HQ) standardizes its business processes in all branches. A workflow template is defined to handle customer problems. When the staff in a branch encounters a problem, a workflow instance is initiated from the template and executed until completion.

The PN specification of the initial template is given in Fig. 2(a). A problem goes through two stages: problem solving and on-site realization. Problem solving involves several tasks, included in a dashed box. When opening a case, the staff in the branch reports the case to the HQ. When closing the case, the staff archives the related documents. The HQ manages all instances related to the problem handling process.

In response to business needs, the HQ may decide to change the problem handling template. The new template (Fig. 2(b)) differs from the original one in two points: a) "reporting" and "problem solving" become independent; b) "on site realization" can fail, in that case "problem solving" procedure restarts.

At Petri net level, we can observe that transition `Report` is causally-connected to `ProductChange` in Fig. 2(a), while is not in Fig. 2(b), and that a new transition has been added in Fig. 2(b) (`RealizationRejected`, which is in free-choice conflict with `OnSiteRealization`).

When using reflective PN, the evolutionary schema is completely different. The new workflow template is not passed as input to the staff of the company branches, but it results from applying an evolutionary strategy to a workflow instance belonging to the current template. The initial base-level PN is assumed a *free-choice WF-net*. No details related to the workflow dynamics are hard-wired in the base-level net. Evolution is delegated to the meta-program, that acts on the WF-net reification. The meta-program is activated when an evolutionary signal is sent in by HQ, or some anomaly (e.g., a deadlock) is revealed by introspection. Introspection is also used to discriminate whether evolutionary commands can be safely applied to the current workflow instance, or they have to be discarded.



(a) initial situation (b) change adjacency preserving (c) change not preserving adjacency

Fig. 3. Definition 4 illustrated.

Fig. 2 depicts the following situation: a workflow instance running on the initial template (Fig. 2(a)) has received a message from HQ. At the current state "solution design" (a sub-task of "problem solving") and "report" are pending, whereas a number of tasks (e.g., "analysis" and "case opening") have finished. The meta-program in that case successfully operates a change on the old template's instance, once verified that all paths to any pending tasks are composed by bypassable tasks only. The workflow instance transferred to the new template is illustrated in Fig. 2(b)). One might think of this approach as instance-based, rather than template-based. In truth it covers both: if the evolutionary commands are broadcasted to workflow instances we fall in the latter scheme.

The evolutionary strategy relies upon the definition of *adjacency preserving* node, more general than the *unchanged* node notion used in Qiu and Wong [2007]. It is inspired by van der Aalst's general concept that a workflow change must preserve the inheritance relationship between old and new templates (van der Aalst and Basten [2002]).

Let us introduce some *notations*. Symbols x,$\overline{x}$ will be used to denote the same node before and after change, respectively: x belongs to a WF-net PN, $\overline{x}$ belongs to the net PN' resulting from change. NEW_P, DEL_P, NEW_T, DEL_T, and NEW_A, DEL_A, denote the base level places/transitions/arcs to be added and removed, respectively; $DEL\_N = DEL\_P \cup DEL\_T$, $NEW\_N = NEW\_P \cup NEW\_T$. Nodes/arcs before change are denoted by OLD_N, OLD_A. Finally, NO_ADJ, NO_BYPS denote the set of nodes not preserving adjacency and non-bypassable, respectively ($NO\_ADJ \subseteq NO\_BYPS$).

Definition 4. (adjacency preserving transition)

Let $\mathcal{A}_t = {}^\bullet({}^\bullet t \cup t^\bullet) \cup ({}^\bullet t \cup t^\bullet)^\bullet$. t is adjacency preserving if and only if $\mathcal{A}_{\overline{t}} \cap OLD\_N = \mathcal{A}_t$ and there exist a bijection $\varphi : {}^\bullet t \cup t^\bullet \rightarrow {}^\bullet \overline{t} \cup \overline{t}^\bullet$ such that $\forall x \in \mathcal{A}_t \forall y \in {}^\bullet t \cup t^\bullet$, $y \in {}^\bullet x \Leftrightarrow \varphi(y) \in {}^\bullet \overline{x}$ and $y \in x^\bullet \Leftrightarrow \varphi(y) \in \overline{x}^\bullet$

If t is adjacency preserving then all its causality/conflict relationships to adjacent tasks are maintained. A case where Def. 4 holds, and another one where it does not, are illustrated in Fig. 3 (the black bar denotes a new task).

Checking definition 4 is computationally expensive. However, if useless changes are forbidden, e.g., "deleting a given place p, then adding p' inheriting p's connections", or "adding an arc $\langle p, t\rangle$, then deleting p or t", then check's complexity is greatly reduced. Lemma 5 states some rules for identifying a *superset* N_a of nodes not preserving adjacency that can be easily translated to an efficient meta-routine. In most practical cases $N_a = NO\_ADJ$.

Lemma 5. Consider set N_a, built as follows

$p \in DEL\_P \Rightarrow {}^\bullet p \cup p^\bullet \subseteq N_a$

$t \in DEL\_T \Rightarrow {}^\bullet({}^\bullet t) \cup ({}^\bullet t)^\bullet \subseteq N_a$

$\langle p, t\rangle \in DEL\_A \vee \langle t, p\rangle \in DEL\_A \Rightarrow {}^\bullet p \cup p^\bullet \subseteq N_a$

$\langle p, t\rangle \in NEW\_A \wedge t \in OLD\_N \Rightarrow \{t\} \cup A \subseteq N_a$,

where $A = {}^\bullet p \cup p^\bullet$ if $p \in OLD\_N$, else $A = \emptyset$.

Then $NO\_ADJ \subseteq N_a$

The evolutionary meta-program corresponds to the CSP-like code in listing 1. The meta-program is activated at any transition of state on the current workflow instance (shift-up), reacting to three different types of events. In the case of deadlock, a signal is sent to HQ, represented by a CSP process identifier. If the current instance has finished, and a "new instance" message is received, the workflow is activated. Instead if there is an incoming evolutionary message from HQ, the evolutionary strategy starts running.

Just after an evolutionary signal, HQ communicates the workflow nodes/connections to be removed/added. For the sake of simplicity we assume that change can only involve workflow topology. The (super)set of non-bypassable nodes is then computed.

After operating the evolutionary commands on the current workflow reification, definition 2 and free-choiceness are checked out on the newly changed reification. Following, the strategy checks by reification introspection whether the suggested workflow change might cause a deadlock, or there might be any non-bypassable tasks causally-connected to an old task which is currently pending. In either case, a restart procedure takes the workflow reification back to the state before strategy's activation. Otherwise, change is reflected to the base-level (shift-down). The scheme just described might be adopted for a wide class of evolutionary patterns.

```
*[
   VAR p, t, n : NODE;
   VAR NEW_P,NEW_T,OLD_N,DEL_N,
       NO_BYPS : SET(NODE);
   VAR NEW_A,DEL_A: SET(ARC);
   HQ ? change-msg() → [
    //receiving evolutionary commands
    HQ ? NEW_P; HQ ? NEW_T; HQ ? NEW_A;
    HQ ? DEL_A; HQ ? DEL_N;
    //computing WF-net reification
    OLD_N = ReifiedNodes();
    //computing non-bypassable tasks
    NO_BYPS = ccTo(notAdjPres());
    //changing reification
    newNode(NEW_P+NEW_T); newArc(NEW_A);
    deleteArc(DEL_A); delNode(DEL_N);
    //checking WF-net well-formedness
    checkWfNet(); checkFc();
    /*there could be a deadlock, or
      a non-bypassable task is causally
      connected to a pending one ...*/
    !exists t in Tran, enab(t) or
    (exists t in Tran * OLD_N, enab(t)
     and !isEmpty(ccBy(t)* NO_BYPS))
      → [restart()] //doing no change
    shiftDown() //reflecting change
    ]
  □
    #end=0 and !exists t in Tran,
     enab(t)
      → [HQ ! notify-deadlock()]
  □
    #end=1; HQ ? newInstance-msg()
      → [flush(end); incMark(begin)]
]
```

Listing 1. workflow evolutionary strategy

```
[
  VAR t,p,t1 : NODE;
  VAR FC : SET(NODE);
  *(⟨ p,t⟩ in NEW_A + DEL_A)
    [
     exists(p) and exists(t) → [
      *(t1 in post(p) \ FC)  [
       t1 <> t and pre(t) <> pre(t1)
              → [ restart()]
       FC = FC + post(p) ]
       ]
     ]
]
```

Listing 2. piece of code checking free-choiceness

Language built-ins and routine calls are in bold. The *NODE* type represents a (logically unbounded) recipient of base-level nodes, and is partitioned into *PLACE* and *TRAN* subtypes . A particular version of CSP repetitive command is used: letting set E be *finite*, `*(e in E)[ command ]` makes `command` to be executed iteratively for each $e \in E$. Note the overloading of operator '*', which is used also to denote the set intersection. The `exists` quantifier is used to check whether a net element is currently reified. The built-in routine **ReifNodes** computes the nodes belonging to the current base-level reification. The routine **notAdjPres** initializes the set of non-bypassable nodes, according to lemma 5. The routines **ccTo** and **ccBy** compute the set of nodes that routine's argument is causally connected to, and that are causally connected to routine's argument, respectively. Listing 2 expands the routine checking preservation of base-level's free-choiceness (**checkFc**). Let us explain how the strategy works considering again Fig. 2. After receiving evolutionary commands:

-NEW_PLACE={};DEL_NODE={}

-NEW_TRAN={RealizationRejected};

-DEL_ARC=$\{\langle p13, \texttt{ProductChange}\rangle\}$;

-NEW_ARC=$\{\langle p6, \texttt{RealizationRejected}\rangle,$

$\langle p13, \texttt{Archiving}\rangle, \langle \texttt{RealizationRejected}, p5\rangle\}$.

The non-bypassable tasks come to be: `Report`, `Archiving`, `ProductChange`, `OnSiteRealization`, `CaseClosure`. In the new workflow tasks `Report` and `ProductChange` are pending (enabled) in the current marking $M : \{p_{11}, p_{14}\}$ of the net (PN') in Fig. 2(b). All old completed tasks that are causally connected to one of them can be bypassed, so the new workflow has not to be restarted from scratch, saving a lot of work.

The approach just described ensures a dependable evolution of workflows, while being enough flexible. We do not intend to propose a general solution to the problem addressed in Qiu and Wong [2007]. Better policies do probably exist. Rather, we aim at showing that the approach merging consolidated reflection concepts to classical PN techniques can suitably address the criticisms of dynamic workflow change.

The base-level PN, which is guaranteed to be a free-choice WF-net during its evolution, may be analyzed using different polynomial techniques. Structural techniques, in particular, are elegant and very efficient, but in general they are highly affected by model complexity. The separation between evolutionary and functional aspects encourages their usage.

By operating the structural algorithms of GreatSPN tool (Chiola et al. [1995]), it is possible to discover that both models in Fig. 2 are covered by *place-invariants*. Thereby a lot of interesting properties descend: in particular boundedness and liveness, i.e., workflow soundness.

5.1 Counter example

Assume that evolution takes place when the only pending task is `OnSiteRealization` (i.e., consider as current marking in Fig. 2(a) $M' : \{p_6\}$), that means, among other, tasks `ProductChange`, `VersionMerging` and `Report` have finished: change in that case is discarded after verifying that there are some non-bypassable tasks causally connected to the only pending one.

If the suggested change were carried out (reflected) without any consistency control, a deadlock would be eventually entered (state $\{p_8\}$) after the process continues

running on the modified template. The problem is that M' is not a reachable state of $(PN'; \{begin\})$, but reachability is NP-complete in live and safe free-choice Petri nets, so it would not make sense checking reachability at meta-program level.

6. CONCLUSION

Covering the intrinsic dynamism of modern processes has been widely recognized as a challenge by designers of workflow management systems. PN are a central model of workflows, but traditionally they have a fixed structure. We have proposed and discussed the adoption of reflective PN as a formal model for designing sound dynamic workflows. A clean separation between the current behavior and the evolution of a workflow, and the use of efficient PN structural techniques, make it possible to check basic workflow properties while evolution is in progress. As an application, an algorithm is delivered to soundly transferring workflow instances from an old to a new template. Ongoing research is in two directions: i) integrating the approach into the GreatSPN package, ii) using a high-level PN class also for the base-level of the reflective model, to incorporate both resources and data in the process description.

REFERENCES

Alessandra Agostini and Giorgio De Michelis. A Light Workflow Management System Using Simple Process Models. *CSCW*, 9(3-4):335–363, August 2000.

Eric Badouel and Javier Oliver. Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes within Workflow Systems. IRISA Research Report PI-1163, January 1998.

Lawrence Cabac, Michael Duvignau, Daniel Moldt, and Heiko Rölke. Modeling Dynamic Architectures Using Nets-Within-Nets. In Gianfranco Ciardo and Philippe Darondeau, editors, *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005)*, LNCS 3536, pages 148–167, Miami, FL, USA, June 2005. Springer.

Lorenzo Capra and Walter Cazzola. A Petri-Net Based Reflective Framework. In Farhad Arbab and Marjan Sirjani, editors, *Proceedings of the IPM International Workshop on Foundations of Software Engineering (FSEN'05)*, ENTCS 159, pages 41–59, Tehran, Iran, on 1st-3rd of October 2005. Elsevier.

Giovanni Chiola, Giuliana Franceschinis, Rossano Gaeta, and Marina Ribaudo. GreatSPN 1.7: GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1-2):47–68, November 1995.

J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.

Clerance Ellis and Karim Keddara. ML-DEWS: Modeling Language to Support Dynamic Evolution within Workflow Systems. *CSCW*, 9(3-4):293–333, August 2000.

Awatef Hicheur, Kamel Barkaoui, and Noura Boudiaf. Modeling Workflows with Recursive ECATNets. In *Prooceedings of SYNACS'06*, pages 389–398, Timişoara, Romania, September 2006. IEEE CS.

Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prantice Hall, 1985.

Kathrin Hoffmann, Hartmut Ehrig, and Till Mossakowski. High-Level Nets with Nets and Rules as Tokens. In Gianfranco Ciardo and Philippe Darondeau, editors, *Proceedings of ICATPN 2005*, LNCS 3536, pages 268–288, Miami, FL, USA, June 2005. Springer.

Kurt Jensen and Grzegorz Rozenberg, editors. *High-Level Petri Nets: Theory and Applications*. Springer-Verlag, 1991.

Pattie Maes. Concepts and Experiments in Computational Reflection. In N. K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, FL, USA, October 1987. ACM.

Zhi-Ming Qiu and Yoke San Wong. Dynamic Workflow Change in PDM Systems. *Computers in Industry*, 58 (5):453–463, June 2007.

Manfred Reichert and Peter Dadam. ADEPTflex - Supporting Dynamic Changes in Workflow Management Systems without Losing Control. *Journal of Intelligent Information Systems*, 10((I2)):93–129, 1998.

Wolfang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer-Verlag, 1991.

Khodakaram Salimifard and Mike B. Wright. Petri Net-Based Modeling of Workflow Systems: An Overview. *European Journal of Operational Research*, 134(3):664–676, November 2001.

Wil M.P̃. van der Aalst. Structural Characterizations of Sound Workflow Nets. Computing Science Reports 96/23, Eindhoven University of Technology, 1996.

Wil M.P̃. van der Aalst and Twan Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 270(1-2):125–203, January 2002.

Wil M.P̃. van der Aalst and S.Jablonski. Dealing with Workflow Change: Identification of Issues and Solutions. *International Journal of Computer Systems, Science, and Engineering*, 15(5):267–276, September 2000.