Exterminating the Dynamic Change Bug

A Concrete Approach to Support Workflow Change

W.M.P. van der Aalst

Eindhoven University of Technology, Faculty of Technology and Management, Department of Information and Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
w.m.p.v.d.aalst@tm.tue.nl

Abstract. Adaptability has become one of the major research topics in the area of workflow management. Today's workflow management systems have problems dealing with both ad-hoc changes and evolutionary changes. As a result, the workflow management system is not used to support dynamically changing workflow processes or the workflow process is supported in a rigid manner, i.e., changes are not allowed or handled outside of the workflow management system. In this paper, we focus on a notorious problem caused by workflow change: the "dynamic change bug" (Ellis et al., 1995). The dynamic change bug refers to errors introduced by migrating a case (i.e., a process instance) from the old process definition to the new one. A transfer from the old process to the new process can lead to duplication of work, skipping of tasks, deadlocks, and livelocks. This paper describes an approach for calculating a safe change region. If a case is in such a change region, the transfer is postponed.

Keywords: Workflow management, Workflow change, Dynamic change, and Petri nets.

1 Introduction

Workflow management technology aims at the automated support and coordination of business processes to reduce costs and flow times, and increase quality of service and productivity. A critical challenge for workflow management systems is their ability to respond effectively to changes. Changes may range from ad-hoc modifications of the process for a single customer to a complete restructuring of the workflow process to improve efficiency. Today's workflow management systems are ill suited to dealing with change. They typically support a more or less idealized version of the preferred process. However, the real run-time process is often much more variable than the process specified at design-time. The only way to handle changes is to go behind the system's back. If users are forced to bypass the workflow management system quite frequently, the system is more a liability than an asset. Therefore, we take up the challenge to find techniques to add flexibility without loosing the support provided by today's systems.

Typically, there are two types of changes: (1) ad-hoc changes and (2) evolutionary changes. Ad-hoc changes are handled on a case-by-case basis. In order to provide customer specific solutions or to handle rare events, the process is adapted for a single case or a limited group of cases. Evolutionary change is often the result of reengineering efforts. The process is changed to improve responsiveness to the customer or to improve the efficiency (do more with less). The trend is towards an increasingly dynamic situation where both ad-hoc and evolutionary changes are needed to improve customer service and reduce costs. In this paper, we restrict ourselves to evolutionary change. In fact, ad-hoc change is partially handled by at least two existing workflow management systems: InConcert (Tibco/InConcert) and Ensemble (FileNet).

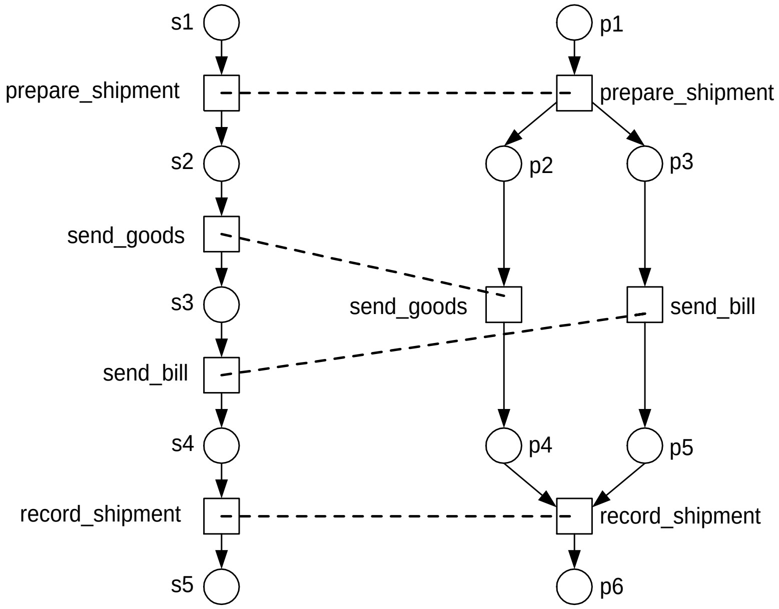

Fig. 1. The dynamic change bug.

The term *dynamic change* refers to the problem of handling old cases in a new process, e.g., how to transfer cases to a new version of the process. The dynamic change problem which was first mentioned by Ellis, Keddara, and Rozenberg (Ellis et al., 1995) in 1995. To discuss this problem we use the two Petri nets shown in Figure 1. For an introduction to Petri nets (Reisig and Rozenberg, 1998) we refer to Appendix A. If the sequential workflow process (left) is changed into the workflow process where tasks *send_goods* and *send_bill* can be executed in parallel (right) there are no problems, i.e., it is always possible to transfer a case from the left to the right. The sequential process has five possible states and each of these states corresponds to a state in the parallel process. For example, the state with a token in *s3* is mapped onto the state with a token in *p3* and *p4*. In both cases, tasks *prepare_shipment* and *send_goods* have been executed and *send_bill* and *record_shipment* still need to be executed. Now consider the situation where the parallel process is changed into the sequential one, i.e., a case is moved from the right-hand-side process to the left-hand-side process. For most of the states of the right-hand-side process this is no problem, e.g., a token in *p1* is moved to *s1*, a token in *p3* and a token in *p4* are mapped onto one token in *s3*, and a token in *p4* and a token in *p5* are mapped onto one token in *s4*. However, the state with a token in both *p2* and *p5* (*prepare_shipment* and *send_bill* have been executed) causes problems because there is no corresponding state in the sequential process (it is not possible to execute *send_bill* before *send_goods*). If the case is moved to place *s2* or place *s3*, task *send_bill* is executed twice. If the case is moved to place *s4* or place *s3*, task *send_goods* is not executed at all. The example in Figure 1 shows that it is not straightforward to migrate old cases to the new process after a change.

The problem illustrated in Figure 1 is a result of reducing the degree of parallelism by making the process sequential. Similar problems occur when the order of tasks is changed, e.g., two sequential tasks are swapped. Extending the workflow with new tasks, removing parts, or aggregating a group of tasks into a single task may result in similar problems. When changing the workflow on-the-fly, i.e., running cases are transferred to the new process definition, the dynamic change bug is likely to occur. Therefore, the problem is very relevant for a workflow management system truly supporting adaptive workflow. Today's workflow management systems are not able to handle this problem. These systems use a *versioning mechanism*, i.e., every change leads to a new version and each case refers to the appropriate version. If a case starts using a version of the process, it will continue to use this version. The versioning mechanism may be suitable in some situations. An administrative process with a short flow time is a good candidate for a versioning mechanism. However, there are many situations where the mechanism is not appropriate. If a case has a long flow time, then it is often not acceptable to handle existing cases this way. Consider for example a process for handling mortgage loans. Mortgages typically have a duration of 20 to 30 years. If the process changes every month, this would lead to hundreds of different versions running in parallel. To reduce cost and to keep the processes manageable, the number of active versions (i.e., versions still used by cases) should be kept to a minimum. Also for processes with a shorter flow time, it may be undesirable to have many versions running in parallel. In fact, there may be legal reasons (e.g., starting from January 2001 a new step in the process is mandatory), forcing the transfer of cases to the new process. Unfortunately, problems such as the one illustrated by Figure 1 make a direct transfer hazardous. Without exterminating the *dynamic change bug* the next generation of workflow management systems will be unable to truly support adaptive workflow.

To exterminate the dynamic change bug, we propose an approach that automatically calculates the change region. The change region is determined by comparing the old and the new process and extending the regions that have changed with regions that are affected by the change. Due to the possibly complex mixture of different routing constructs (choice, synchronization, iteration, etc.), it is far from trivial to compute the regions that are affected. If a case is in the change region, it cannot be transferred. The transfer is delayed until the change region is empty. We will show that postponing the migration of a case until the change region is empty, results in the correct execution of the case. For a short period (depending on the change region), there are multiple versions but as soon as a transfer is safe the case is handled according to the new process definition.

The approach differs from existing approaches (Aalst and Basten, 2001; Aalst et al., 2000b; Aalst et al., 2000a; Casati et al., 1998; Reichert and Dadam, 1998; Ellis et al., 1995; Ellis and Keddara, 2000a; Ellis and Keddara, 2000b; Michelis and Ellis, 1998; Keddara, 1999; Joeris and Herzog, 1998; Agostini and Michelis, 2000; Sadiq et al., 2000; Vossen and Weske, 1999; Weske, 2000) in the sense that no local transformation rules are assumed and that the calculation of the change region is based on the structure of the workflow graph. Note that we restrict ourselves to logical errors in the control-flow resulting from change. Clearly, there are other types of constraint violations. Moreover, change can also result in resource or data conflicts. These problems are outside the scope of this paper.

The remainder of this paper is organized as follows. First, we introduce the basic concepts and the techniques we are going to use. The approach presented in this paper is based on a special subclass of Petri nets (WF-nets) and a notion of correctness named soundness (Aalst, 1998b; Aalst, 2000). In Section 3, we clearly define to problem (the dynamic change bug) and give a number of dynamic change examples. Then, we present the algorithm to calculate the change region. In Section 5, we compare this approach with other approaches addressing the dynamic change problem. Finally, we summarize the results presented and conclude with our future plans.

2 Preliminaries

This section introduces the basic concepts, definitions, and techniques used to tackle the dynamic change bug. For a more elaborate discussion on these topics, we refer to (Aalst, 1998b; Aalst, 2000; Ellis et al., 1995; Ellis and Nutt, 1993). For an introduction to Petri nets we refer to (Desel and Esparza, 1995; Murata, 1989; Reisig and Rozenberg, 1998) and the appendix.

2.1 Workflow process definitions

The term workflow management (Koulopoulos, 1995; Lawrence, 1997; Jablonski and Bussler, 1996) refers to the domain which focuses on the logistics of business processes. There are also people who use the term *office logistics*. The ultimate goal of workflow management is to make sure that the proper activities are executed by the right person at the right time. Although it is possible to do workflow management without using a workflow management system, most people associate workflow management with workflow management *systems*. The *Workflow Management Coalition* (WfMC) defines a workflow management system as follows (WFMC, 1996): *A system that completely defines, manages, and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic*. Other terms to characterize a workflow management system are: ‘business operating system’, ‘workflow manager’, ‘case manager’ and ‘logistic control system’.

Workflows are *case-based*, i.e., every piece of work is executed for a specific *case*. Examples of cases are a mortgage, an insurance claim, a tax declaration, an order, or a request for information. Cases are often generated by an external customer. However, it is also possible that a case is generated by another department within the same organization (internal customer). The goal of workflow management is to handle cases as efficiently and effectively as possible. A workflow process is designed to handle similar cases. Cases are handled by executing *tasks* in a specific order. The *workflow process definition* specifies which tasks need to be executed and in what order. Alternative terms for workflow process definition are: 'procedure', 'flow diagram' and 'routing definition'. Since tasks are executed in a specific order, it is useful to identify *conditions* which correspond to causal dependencies between tasks. A condition holds or does not hold (true or false). Each task has pre- and postconditions: the preconditions should hold before the task is executed, and the postconditions should hold after execution of the task. Many cases can be handled by following the same workflow process definition. As a result, the same task has to be executed for many cases. A task that needs to be executed for a specific case is called a *work item*. An example of a work item is: execute task 'send refund form to customer' for case 'complaint sent by customer Baker'. Most work items are executed by a *resource*. A resource is either a machine (e.g., a printer or a fax) or a person (participant, worker, employee). To facilitate the allocation of work items to resources, resources are grouped into classes. A *resource class* is a group of resources with similar characteristics. There may be many resources in the same class and a resource may be a member of multiple resource classes. If a resource class is based on the capabilities (i.e., functional requirements) of its members, it is called a *role*. If the classification is based on the structure of the organization, such a resource class is called an *organizational unit*. A work item which is being executed by a specific resource is called an *activity*.

Of all workflow perspectives (e.g., control-flow, data, organization, task, operation) (Jablonski and Bussler, 1996), the control-flow perspective is the most prominent one, because the core of any workflow system is formed by the processes it supports. In the control-flow dimension building blocks such as the AND-split, AND-join, OR-split, and OR-join are used to model sequential, conditional, parallel and iterative routing (WFMC, 1996). Clearly, a Petri net can be used to specify the routing of cases. See Appendix A for an introduction to Petri nets. *Tasks* are modeled by transitions and causal dependencies are modeled by places and arcs. In fact, a place corresponds to a *condition* which can be used as pre- and/or post-condition for tasks. An AND-split corresponds to a transition with two or more output places, and an AND-join corresponds to a transition with two or more input places. OR-splits/OR-joins correspond to places with multiple outgoing/ingoing arcs. Moreover, in (Aalst, 1998a) it is shown that the Petri net approach also allows for useful routing constructs absent in many workflow management systems.

A Petri net which models the control-flow dimension of a workflow, is called a *WorkFlow net* (WF-net). It should be noted that a WF-net specifies the dynamic behavior of a single case in isolation.

Definition 1 (WF-net). *A Petri net $PN = (P, T, F)$ is a WF-net (WorkFlow net) if and only if:*

(i) There is one source place $i \in P$ such that $\bullet i = \emptyset$.
(ii) There is one sink place $o \in P$ such that $o\bullet = \emptyset$.
(iii) Every node $x \in P \cup T$ is on a path from i to o.

A WF-net has one input place (i) and one output place (o) because any case handled by the procedure represented by the WF-net is created when it enters the workflow management system and is deleted once it is completely handled by the workflow management system, i.e., the WF-net specifies the life-cycle of a case. The third requirement in Definition 1 has been added to avoid 'dangling tasks and/or conditions', i.e., tasks and conditions which do not contribute to the processing of cases. If there is no confusion possible we will use i and o to denote the input place and output place of a WF-net. If confusion is possible, we add a subscript referring to the proper WF-net, i.e., i_{PN} and o_{PN} denote the input place and output place of the WF-net PN.

Given the definition of a WF-net it is easy derive the following properties.

Proposition 1 (Properties of WF-nets). *Let $PN = (P, T, F)$ be Petri net.*

- *If PN is WF-net with source place i, then for any place $p \in P$: $\bullet p \neq \emptyset$ or $p = i$, i.e., i is the only source place.*
- *If PN is WF-net with sink place o, then for any place $p \in P$: $p\bullet \neq \emptyset$ or $p = o$, i.e., o is the only sink place.*
- *If PN is a WF-net and we add a transition t^* to PN which connects sink place o with source place i (i.e., $\bullet t^* = \{o\}$ and $t^*\bullet = \{i\}$), then the resulting Petri net is strongly connected.*
- *If PN has a source place i and a sink place o and adding a transition t^* which connects sink place o with source place i yields a strongly connected net, then every node $x \in P \cup T$ is on a path from i to o in PN and PN is a WF-net.*

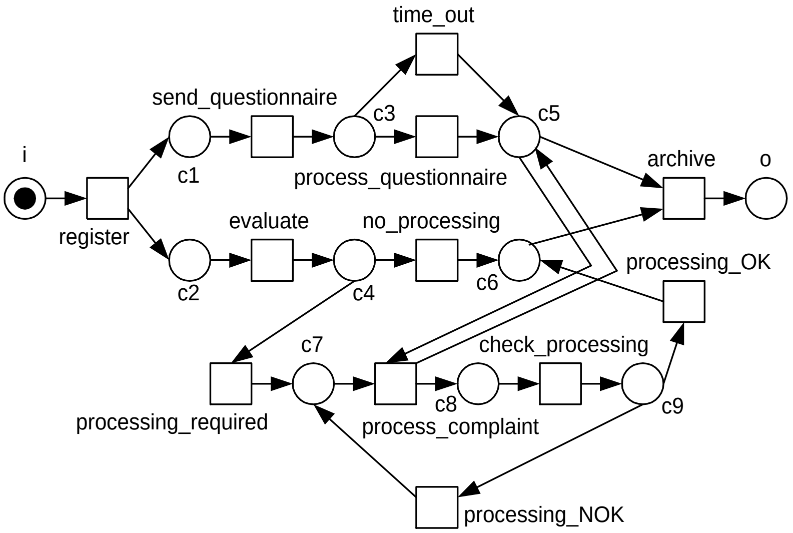

Fig. 2. A WF-net for the processing of complaints.

Figure 2 shows a WF-net which models the processing of complaints. First the complaint is registered (task *register*), then in parallel a questionnaire is sent to the complainant (task *send_questionnaire*) and the complaint is evaluated (task *evaluate*). If the complainant returns the questionnaire within two weeks, the task *process_questionnaire* is executed. If the questionnaire is not returned within two weeks, the result of the questionnaire is discarded (task *time_out*). Based on the result of the evaluation, the complaint is processed or not. The actual processing of the complaint (task *process_complaint*) is delayed until condition *c5* is satisfied, i.e., the questionnaire is processed or a time-out has occurred. The processing of the complaint is checked via task *check_processing*. Finally, task *archive* is executed. Note that sequential, conditional, parallel and iterative routing are present in this example.

The WF-net shown in Figure 2 clearly illustrates that we focus on the control-flow dimension. We abstract from resources, applications, and technical platforms. Moreover, we also abstract from *case variables* and *triggers*. Case variables are used to resolve choices (OR-split), i.e., the choice between *processing_required* and *no_processing* is (partially) based on case variables set during the execution of task *evaluate*. The choice between *processing_OK* and *processing_NOK* is resolved by testing case variables set by *check_processing*. In the WF-net we abstract from case variables by introducing non-deterministic choices in the Petri-net. If we don't abstract from this information, we would have to model the (unknown) behavior of the applications used in each of the tasks and analysis would become intractable. In Figure 2 we have not indicated that *time_out* and *process_questionnaire* require triggers. Task *time_out* requires a time trigger ('two weeks have passed') and *process_questionnaire* requires a message trigger ('the questionnaire has been returned'). A trigger can be seen as an additional condition which needs to be satisfied. In the remainder of this paper, we abstract from these trigger conditions. We assume that the environment behaves fairly, i.e., the liveness of a transition is not hindered by the continuous absence of a specific trigger. As a result, every trigger condition will be satisfied eventually.

2.2 Soundness property

In this subsection we summarize some of the basic results for WF-nets presented in (Aalst, 2000). The remainder of this paper will build on these results.

The three requirements stated in Definition 1 can be verified statically, i.e., they only relate to the structure of the Petri net. However, there is another requirement which should be satisfied:

> *For any case, the procedure will terminate eventually and the moment the procedure terminates there is a token in place o and all the other places are empty.*

Moreover, there should be no dead tasks, i.e., it should be possible to execute an arbitrary task by following the appropriate route though the WF-net, and the WF-net should be safe. These additional requirements for WF-nets correspond to the so-called *soundness property*.

Definition 2 (Sound). *A procedure modeled by a WF-net* $PN = (P, T, F)$ *is sound if and only if:*

(i) For every state M reachable from state i, there exists a firing sequence leading from state M to state o. Formally:[1]

$$\forall_M (i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o)$$

(ii) State o is the only state reachable from state i with at least one token in place o. Formally:

$$\forall_M (i \xrightarrow{*} M \ \wedge \ M \geq o) \Rightarrow (M = o)$$

(iii) There are no dead transitions in (PN, i). Formally:

$$\forall_{t \in T} \ \exists_{M,M'} \ i \xrightarrow{*} M \xrightarrow{t} M'$$

(iv) (PN, i) is safe.

Note that the soundness property relates to the dynamics of a WF-net. The first requirement in Definition 2 states that starting from the initial state (state *i*), it is always possible to reach the state with one token in place *o* (state *o*). If we assume a strong notion of fairness, then the first requirement implies that eventually state o is reached. Strong fairness means that in every infinite firing sequence, each transition fires infinitely often. The fairness assumption is reasonable in the context of workflow management: All choices are made (implicitly or explicitly) by applications, humans or external actors. Clearly, they should not introduce an infinite loop. Note that the traditional notions of fairness (i.e., weaker forms of fairness with just local conditions, e.g., if a transition is enabled infinitely often, it will fire eventually) are not sufficient. See (Aalst, 1998b; Kindler and Aalst, 1999) for more details. The second requirement states that the moment a token is put in place *o*, all the other places should be empty. Sometimes the term *proper termination* is used to describe the first two requirements (Gostellow et al., 1972). The third requirement states that there are no dead transitions (tasks) in the initial state *i*. The last requirement states that the WF-net should be safe, i.e., for a single case in isolation, it is not allowed to have multiple tokens in one place.

Figure 2 is an example of a WF-net which is sound. Figure 3 shows a WF-net which is not sound. This WF-net is an attempt to simplify the one shown in Figure 2: The token in *c5* is now actually removed by *process_complaint*, i.e., *process_complaint* does not return the token and, therefore, *archive* no longer needs to remove the remaining token in *c5*. Although this may seem to be a good idea, there are several deficiencies. First of all, tasks may be executed after completion, e.g., after firing *register*, *evaluate*, *no_processing*, and *archive* there is a token in place *o* indicating completion, but at the same time *send_questionnaire* is enabled. Second, there is a potential deadlock: If *processing_NOK* fires, then the WF-net gets stuck in the state with just a token in *c7*. Any attempt to execute task *processing_complaint* multiple times will lead to a deadlock situation. Clearly, the WF-net is not sound.

Given a WF-net $PN = (P, T, F)$, we want to decide whether PN is sound. In (Aalst, 2000) we have shown that soundness corresponds to liveness and boundedness. To link soundness to liveness and boundedness, we define an extended net $\overline{PN} =$

[1] Note that there is an overloading of notation: the symbol *i* is used to denote both the *place i* and the *state* with only one token in place *i* (see Appendix A).

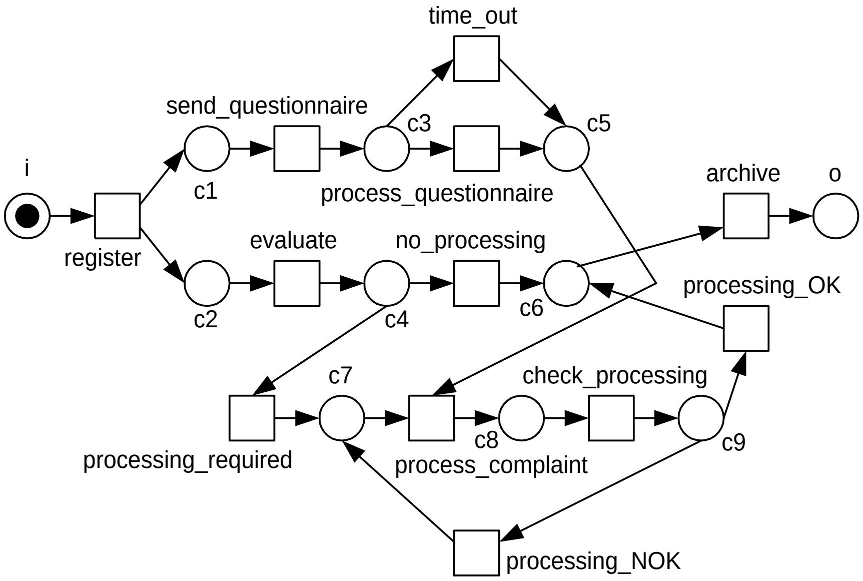

Fig. 3. Another WF-net for the processing of complaints.

$(\overline{P},\overline{T},\overline{F})$. $\overline{PN}$ is the Petri net obtained by adding an extra transition t^* which connects o and i. The extended Petri net $\overline{PN} = (\overline{P},\overline{T},\overline{F})$ is defined as follows: $\overline{P} = P$, $\overline{T} = T \cup \{t^*\}$, and $\overline{F} = F \cup \{\langle o,t^*\rangle, \langle t^*,i\rangle\}$. In the remainder, we will call such an extended net the *short-circuited* net of PN. The short-circuited net allows for the formulation of the following theorem.

Theorem 1. *A WF-net PN is sound if and only if $(\overline{PN}, i)$ is live and safe.*

Proof. See (Aalst, 2000). □

This theorem shows that standard Petri-net-based analysis techniques can be used to verify soundness. The short-circuited version of the WF-net shown in Figure 2 is live and safe. The short-circuited version of the WF-net shown Figure 3 is not live and not safe.

3 Dynamic Change: The Problem

Today's workflow management systems typically support two types of change. The systems aiming at predefined and well-structured workflow processes, often referred to as *production workflow*, support a versioning mechanism (cf. Section 1). Most of the available systems fit into this category (e.g., Staffware, MQ Series workflow, and COSA). Only a few systems support a different form of change by binding private process definitions to cases. The latter class of workflow systems support ad-hoc workflow and examples of such systems are InConcert (InConcert/TIBCO) and Ensemble (Filenet).

The versioning mechanism supported by most of the systems binds each case to a specific version of the workflow. A version itself will never change: Only new versions can be added. Therefore, each case will follow the procedure defined in the corresponding version and will not be influenced by changes during its lifetime. Only new cases benefit from change and typically follow the most recent version of the workflow at the moment of creation.

Systems supporting ad-hoc workflow associate a workflow process definition with each case, i.e., each workflow instance carries its own description. These systems typically allow for limited change, e.g., in InConcert it is possible to remove and/or add tasks in parts of the process which still need to be executed. Clearly, these systems do not support evolutionary changes as described in the introduction.

Both types of change (versioning mechanism and ad-hoc workflow) are quite easy to implement and are not confronted with problems such as the one illustrated by Figure 1. The dynamic change problem, which was first mentioned by Ellis, Keddara, and Rozenberg (Ellis et al., 1995) in 1995, is not addressed at all by these systems. Nevertheless, there is a clear need for mechanisms which allow for the migration of instances (cases) from one process definition to another. There are many examples of workflow processes with a considerable number of instances which have a long flow time. Consider for example mortgage loans which have a life-cycle of decades. In such situations the version mechanism is not acceptable: Too many versions would be active thus resulting in an unmanageable workflow. There may also be legal and economical reasons for the migration of instances (cases) from one process definition to another. If the law changes, some processes may be affected and the organization may be forced to migrate cases, i.e., to handle existing cases the new way. The solution provided by ad-hoc workflow systems is often not acceptable, because every instance needs to be modified by hand and there is no control over the uniformity of the workflow process. Moreover, the solutions provided by systems like InConcert restrict the modeling language to avoid problems such as the one illustrated by Figure 1 (e.g., no iteration). Therefore, we tackle the dynamic change problem using the concepts introduced in the previous section. The notion of soundness will be used as a staring point for the formulating the problem.

In the remainder, we assume that two workflow process definitions are given: (1) the old workflow, i.e., the workflow process definition before the change, and (2) the new workflow, i.e., the workflow after the change. Both workflows are specified in terms of WF-nets. We denote the *old WF-net* and the *new WF-net* as $PN^O = (P^O, T^O, F^O)$ and $PN^N = (P^N, T^N, F^N)$ respectively. We assume that $(P^O \cup P^N) \cap (T^O \cup T^N) = \emptyset$, i.e., no name clashes.

The goal of the approach presented in this paper is to calculate when it is possible to migrate instances (i.e., cases) from the old workflow to the new workflow. For this purpose we need a notion of correctness. In this paper we choose a very pragmatic notion of correctness, a transfer is *valid* if the state of the case *after* migration *could* have been reached from the initial state.

Definition 3 (Valid transfer). *Let $PN^O = (P^O, T^O, F^O)$ and $PN^N = (P^N, T^N, F^N)$ be two sound WF-nets and M a reachable marking of PN^O, i.e., $i_{PN^O} \xrightarrow{*} M$ in PN^O. A transfer $(PN^O, M) \Rightarrow (PN^N, M)$ is valid iff*

(i) for all $p \in P^O : M(p) \geq 1$ implies $p \in P^N$,
(ii) $i_{PN^N} \xrightarrow{*} M$ in PN^N.

The first requirement in Definition 3 states that all marked places should exist in the new workflow, i.e., it is not valid to migrate a case with tokens in places which are removed from the new WF-net. The second requirement states that marking M, i.e., the state of the case to be migrated, is reachable in the new process. The latter property shows that it is not valid to end up in a state not reachable by newly created cases, i.e., cases starting in marking i.

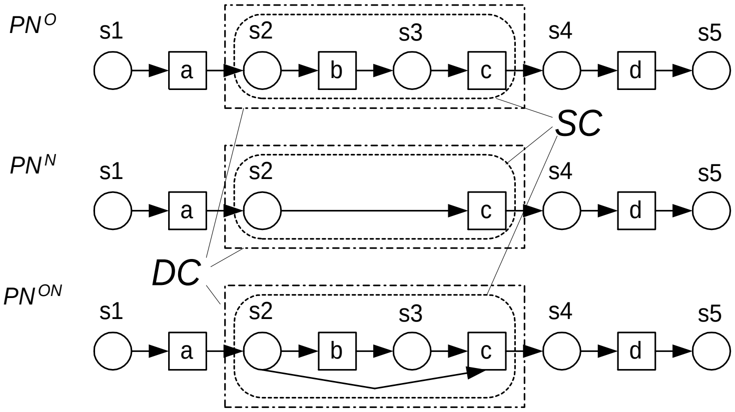

Fig. 4. An old and a new WF-net: $SC = \{s2, s3, b, c\}$ and $DC = \{s2, s3, b, c\}$.

Consider PN^O and PN^N shown in Figure 4. (Ignore PN^{ON}, SC, and DC.) A case marking place $s2$ in PN^O can be migrated to PN^N, i.e., the transfer $(PN^O, s2) \Rightarrow (PN^N, s2)$ is valid. A case marking place $s1$, $s4$, or $s5$ in PN^O can also be migrated to PN^N while satisfying the requirements stated in Definition 3. However, there is no valid transfer for a case marking $s3$. Figure 5 shows two other WF-nets. The old WF-net PN^O uses conditional routing. The new WF-net uses parallel routing. A case marking place $s2$ in PN^O can be migrated to PN^N, i.e., the transfer $(PN^O, s2) \Rightarrow (PN^N, s2)$ is valid. However, there is no valid transfer for a case marking any of the places $s3$, $s4$, $s5$, and $s6$. Consider for example a case with a token in $s3$. If this case is migrated to the new WF-net PN^N, then there is a deadlock. In PN^N the marking $s3$ enables c, but after firing c the case gets stuck in the state just marking $s4$. Note that the state just marking $s3$ is not reachable in the new process.

The notion of validity introduced in Definition 3, guarantees that the essence of soundness is preserved during the migration. After a valid transfer, the case can termi-

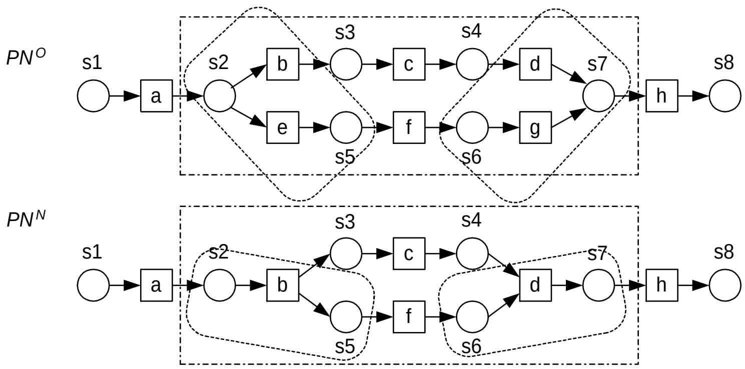

Fig. 5. An old and a new WF-net: $SC = \{s2, s5, s6, s7, b, d, e, g\}$ and $DC = \{s2, s3, s4, s5, s6, s7, b, c, d, e, f, g\}$.

nate in a state just marking the sink place and the moment a case terminates all other places are unmarked.

The goal of this paper is to determine when a transfer is valid. In principle it is possible to calculate whether a transfer is valid using standard techniques such as the reachability graph (cf. (Reisig and Rozenberg, 1998)). However, we are looking for more pragmatic criteria. In practice, a process can have millions of reachable states. To classify these states into valid and invalid requires a complete enumeration of the state space of the old and the new process. Therefore, there are definitely computational problems. Moreover, such a brute-force partitioning of the state space is also very indirect: The partitioning only relates to the graphical workflow model in an indirect manner. Therefore, we pursue a more down-to-earth approach based on *change regions*. The change region is the part of the model which is effected by the change. These change regions are defined in terms of nodes (i.e., tasks, conditions, etc.) in the workflow model instead of states. This allows for more intuitive criteria and facilitates a more realistic implementation.

First we define the *static change region*. The static change region is the set of nodes of both the old and the new process model which are *syntactically* involved in the change.

Definition 4 (Static change region). *Let $PN^O = (P^O, T^O, F^O)$ and $PN^N = (P^N, T^N, F^N)$ be two sound WF-nets. The static change region in the context of a change from PN^O to PN^N is the set $SC = \bigcup_{(x,y)\in X}\{x, y\}$ where $X = (F^O \setminus F^N) \cup (F^N \setminus F^O)$.*

The static change region is calculated by comparing the flow relations of both nets, i.e., all arcs which are removed or added are recorded. The set of all nodes (i.e., places and transitions) linked to an arc which is added or removed constitutes the static change region. Note that the change region consists of nodes in both the old and the new workflow. Definition 4 compares arcs rather than nodes. However, as the following property shows, all nodes added or deleted appear in the static change region.

Property 1. *Let PN^O, PN^N, and SC be defined in Definition 4. $((P^O \cup T^O) \setminus (P^N \cup T^N)) \cup ((P^N \cup T^N) \setminus (P^O \cup T^O)) \subseteq SC$.*

Proof. Let $x \in (P^O \cup T^O) \setminus (P^N \cup T^N)$. PN^O is connected. Therefore, there is a $y \in P^O \cup T^O$ such that $(x, y) \in F^O$ or $(y, x) \in F^O$. Clearly, $(x, y) \notin F^N$ and $(y, x) \notin F^N$ because $x \notin P^N \cup T^N$. Hence, $(x, y) \in F^O \setminus F^N$ or $(y, x) \in F^O \setminus F^N$. Therefore, $x \in SC$. Similarly, it can be shown that $x \in SC$ if $x \in (P^N \cup T^N) \setminus (P^O \cup T^O)$. □

Consider PN^O and PN^N shown in Figure 4. For these two WF-nets $SC = \{s2, s3, b, c\}$. Nodes $s3$ and b have been removed and are part of the change region. Nodes $s2$ and c are also in the static change region because $s2\bullet$ and $\bullet c$ have changed. Projections of the set SC onto P^O and P^N are shown in Figure 4 using dashed ovals. Figure 4 also shows the set SC in the *combined WF-net*. Let PN^O and PN^N be an old and a new WF-net respectively. The combined WF-net is a WF-net denoted as $PN^{ON} = (P^{ON}, T^{ON}, F^{ON})$ and defined as follows: $PN^{ON} = PN^O \cup PN^N$. The union of two Petri nets is defined in Appendix A. It is easy to see that PN^{ON} is a WF-net.

Property 2. *Let PN^O and PN^N be two WF-nets such that $i_{PN^O} = i_{PN^N}$ and $o_{PN^O} = o_{PN^N}$. $PN^{ON} = PN^O \cup PN^N$ is a WF-net.*

Proof. $i_{PN^{ON}} = i_{PN^O} = i_{PN^N}$ is a source place since it cannot have ingoing arcs. There are no other places without any input transitions. Hence $i_{PN^{ON}}$ is a unique source place. Similarly, $o_{PN^{ON}} = o_{PN^O} = o_{PN^N}$ is a unique sink place. Moreover, every node is on a path from $i_{PN^{ON}}$ to $o_{PN^{ON}}$ because this is the case in either PN^O or PN^N. Hence PN^{ON} is a WF-net. □

The combined WF-net contains the union of all nodes and arcs which appear in any of the two WF-nets.

It is important to note that the calculation of the static change region is symmetric, i.e., if the roles of the old and the new WF-net are reversed, the change region does not change. Consider for example Figure 5: $SC = \{s2, s5, s6, s7, b, d, e, g\}$. If the roles of the two nets are reversed, i.e., the parallel routing is changed into a conditional one, then the static change region is still $SC = \{s2, s5, s6, s7, b, d, e, g\}$.

One might think that as long as the static change region is unmarked, a migration of the old WF-net to the new one is valid. Consider for example Figure 6 where tasks c and f are replaced by j and k, and, subsequently, $SC = \{s3, s4, s5, s6, c, f, j, k\}$. Any case marking only places outside SC, can be migrated without any problems. In fact, even for cases marking $s3$, $s4$, $s5$, and $s6$ there is a valid transfer. However, there are situations were the transfer of a case not marking any of the places in the change region is invalid. Consider for example Figure 5 and a case marking place $s3$ in PN^O. This state is reachable from the initial state $s1$ of PN^O and $s3$ is not part of $SC = \{s2, s5, s6, s7, b, d, e, g\}$. Although the case is not marking any of the places in the change region, the transfer is *not* valid. Transferring the token in $s3$ from PN^O to PN^N results in a marking not reachable from the initial state $s1$ of PN^N. This example shows that the static change region is not a good characterization of the part

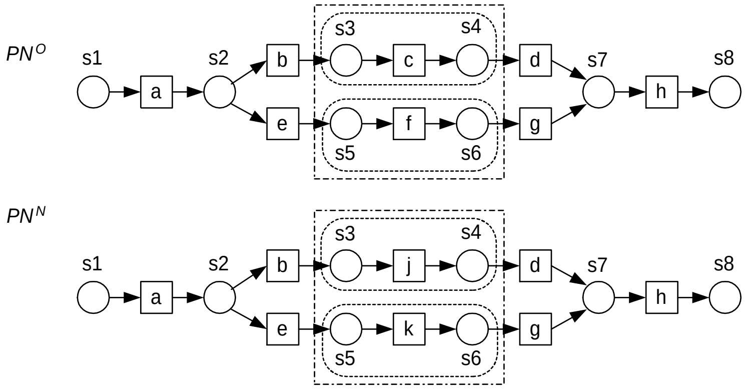

Fig. 6. An old and a new WF-net: $SC = \{s3, s4, s5, s6, c, f, j, k\}$ and $DC = \{s3, s4, s5, s6, c, f, j, k\}$.

of the WF-net which should be unmarked to allow for a valid transfer. This problem is addressed in the next section.

4 Dynamic Change: The Solution

The static change region introduced in the previous section is a very elegant and tangible notion. For readers familiar with the UNIX operating system; the static change region is comparable to the *diff* program which calculates the differences between two UNIX files. It is quite straightforward to build a small application program which calculates the static change region of two workflow process definition specified using a given workflow management system. However, as was illustrated using Figure 5, there are situations where an unmarked static change region does not guarantee a valid transfer. In this section we present an algorithm which calculates a change region which guarantees that any transfer is valid as long as the change region is unmarked. We will use the term *dynamic change region* for this region. We will prove that the dynamic change region provides a *sufficient* condition for validity, i.e., any case not marking the dynamic change region can be transferred without jeopardizing the correctness criteria mentioned. The dynamic change region does not provide a *necessary* condition for validity: this is an inherit consequence of the fact that we want a syntactical criterion rather than a criterion based on the explicit enumeration of the state space.

The calculation of the dynamic change region DC starts from SC and continues to extend this set until certain syntactical requirements are met. The algorithm uses the combined WF-net and forms components (i.e., locally connected change regions) which correspond to sound "sub-WF-nets".

Definition 5 (Dynamic change region). *Let $PN^O = (P^O, T^O, F^O)$ and $PN^N = (P^N, T^N, F^N)$ be two sound WF-nets and let $PN^{ON} = (P^{ON}, T^{ON}, F^{ON})$ be the combined WF-net, i.e., $PN^{ON} = PN^O \cup PN^N$ and $i_{PN^{ON}} = i_{PN^O} = i_{PN^N}$ and*

$o_{PN^{ON}} = o_{PN^O} = o_{PN^N}$. SC is the static change region. The dynamic change region DC is calculated by the following algorithm.

```
Algorithm 1 [Dynamic Change Region Generation Algorithm]
begin
01.  $DC := \emptyset$
02.  $X := SC$
03.  while $DC \neq X$ do
      begin
04.       $DC := X$
05.       partition $X$ into $X_1$, $X_2$, ... ,$X_n$ such that
            (a) $X_i \cap X_j = \emptyset$ for all $1 \leq i < j \leq n$
            (b) $X = \bigcup_{1 \leq i \leq n} X_i$
            (c) $PN^{ON}|_{X_i}$ is connected for all $1 \leq i \leq n$
            (d) $(\bullet X_i) \cap X_j = \emptyset$ and $(X_i \bullet) \cap X_j = \emptyset$ for all $1 \leq i < j \leq n$
06.       for $k := 1..n$ do
07.           for $a \in (X_k)$ do
08.               for $b \in ((X_k) \backslash \{a\})$ do
09.                   for $c \in (P^{ON} \cup T^{ON})$ do
                       begin
10.                           for $(C_1 \in paths(a,c)) \wedge (C_2 \in paths(b,c)) \wedge$
                                $(\alpha(C_1) \cap \alpha(C_2)) = \{c\}$ do
11.                               $X = X \cup \alpha(C_1) \cup \alpha(C_2)$
12.                           for $(C_1 \in paths(c,a)) \wedge (C_2 \in paths(c,b)) \wedge$
                                $(\alpha(C_1) \cap \alpha(C_2)) = \{c\}$ do
13.                               $X = X \cup \alpha(C_1) \cup \alpha(C_2)$
                       end
14.       $X = X \cup (\bigcup_{\substack{x \in X \\ \bullet x \cap X \neq \emptyset}} \bullet x) \cup (\bigcup_{\substack{x \in X \\ x \bullet \cap X \neq \emptyset}} x \bullet)$
      end
15.  output $DC$
end
```

The algorithm initializes X as the set of nodes in the static change region, i.e., $X = SC$. Then using a number of iterations, this set X is extended. During each iteration the set X is partitioned into subsets X_i which correspond to connected components. Note that the projection of a net PN onto a set of nodes X ($PN|_X$) is defined in Appendix A. For each component and each pair of nodes in a component, the algorithm searches for elementary paths which start or end in these two nodes and end or start in a single common node c. Function $paths$ returns the set of all elementary paths between two given nodes, i.e., $paths(a, b)$ is the set of *elementary* paths which start in node a and end in node b (see Appendix A). The alphabet operator α is a function which returns the set of nodes on a given path. If two paths are found which start/end in a and b and end/start in c and only overlap in c, then all nodes on both paths are added to the set X (see lines 11 and 13). If a node x is an element of X and an input (output) node of x is an element of X, then all input (output) nodes $\bullet x$ ($x\bullet$) are also added to X (see line 14 of the algorithm).

The complexity of the straightforward implementation of the algorithm is factorial ($O(n^4(n!)^2)$ for a workflow with n nodes). From a practical point of view, its complexity is acceptable because the algorithm only considers the graph structure of the WF-net. The algorithm does not enumerate all possible states and is executed only once per change, i.e., there is no need to compute the dynamic change region for individual cases. Moreover, a typical workflow consists of less than 50 nodes. Despite its factorial complexity, the Dynamic Change Region Generation Algorithm is tractable for the workflows encountered in practice.

Consider for example Figure 4. The dynamic change region coincides with the static change region, i.e., $DC = SC = \{s2, s3, b, c\}$. The dynamic change region and the static change region also coincide for the two WF-nets shown in Figure 6: $DC = SC = \{s3, s4, s5, s6, c, f, j, k\}$. Figure 5 shows an example of a situation where both regions do not coincide. The static change region $SC = \{s2, s5, s6, s7, b, d, e, g\}$ does not include $s3$, $s4$, c, and f. However, these nodes are influenced by the change. In PN^O only one of the tasks c and f is executed (conditional routing) while in PN^N both tasks are executed (parallel routing). As was indicated before, it is not possible to migrate cases marking $s3$ or $s5$ without resulting in an invalid transfer. Therefore, the dynamic change region includes $s3$, $s4$, c, and f, i.e., $DC = \{s2, s3, s4, s5, s6, s7, b, c, d, e, f, g\}$.

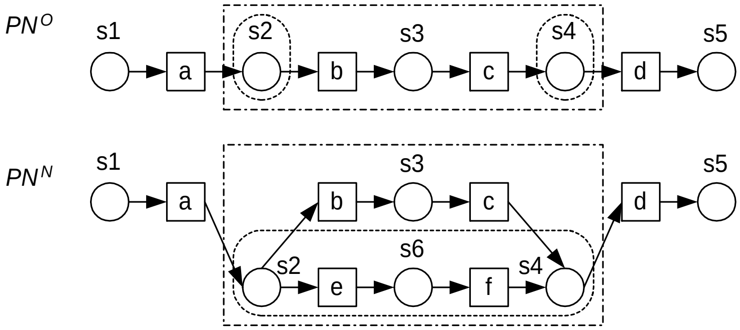

Fig. 7. An old and a new WF-net: $SC = \{s2, s4, s6, e, f\}$ and $DC = \{s2, s3, s4, s6, b, c, e, f\}$.

Figures 7, 8, and 9 show three additional examples. For each example both the dynamic and the static change region are indicated. Figure 7 shows the addition of an alternative branch containing tasks e and f. The static change region $SC = \{s2, s4, s6, e, f\}$ only addresses the places $s2$ and $s4$ in PN^O. The dynamic change region also includes b, $s3$, and c, i.e., $DC = \{s2, s3, s4, s6, b, c, e, f\}$. Figure 8 shows the addition of a parallel branch containing tasks e and f. Place $s3$ is not included in the static change region $SC = \{s6, s7, s8, b, c, e, f\}$. However, it is clear that $s3$ also needs to be included. The transfer of a case in state $s3$ from PN^O to PN^N results in a deadlock. Therefore, place $s3$ is included in the dynamic change region, i.e., $DC = \{s3, s6, s7, s8, b, c, e, f\}$. Figure 9 shows the addition of a feedback loop. This example shows the effect of line 14 of the algorithm: If a node x is an element of X and an input (output) node of x is

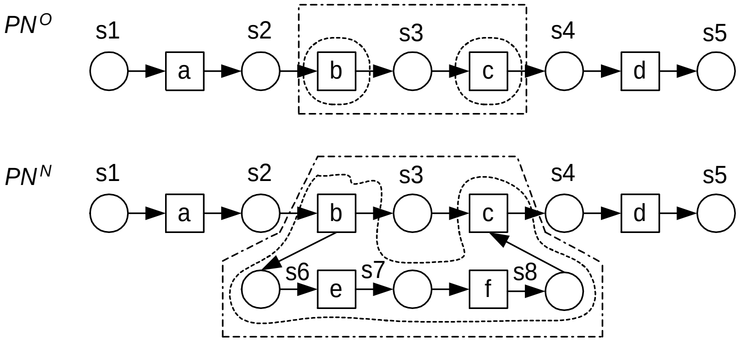

Fig. 8. An old and a new WF-net: $SC = \{s6, s7, s8, b, c, e, f\}$ and $DC = \{s3, s6, s7, s8, b, c, e, f\}$.

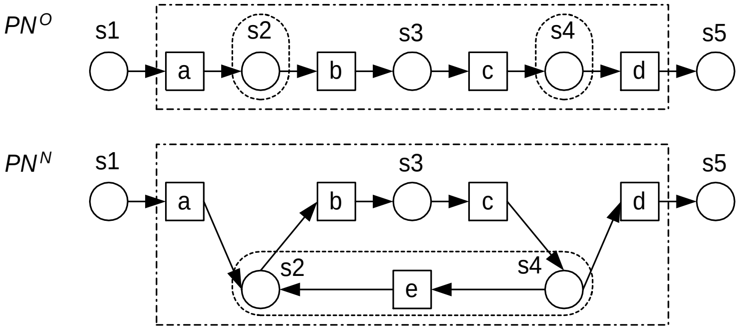

Fig. 9. An old and a new WF-net: $SC = \{s2, s4, e\}$ and $DC = \{s2, s3, s4, a, b, c, d, e\}$.

an element of X, then all input (output) nodes $\bullet x$ ($x\bullet$) are also added to X. Because of this line the tasks a and d are added to the dynamic change region. Note that the dynamic change region $DC = \{s2, s3, s4, a, b, c, d, e\}$ is considerably larger than the static change region $SC = \{s2, s4, e\}$.

The following theorem shows that the dynamic change region calculated by the algorithm can be used to guarantee the validity of transfers, i.e., if only cases outside the dynamic change region are transferred, then any transfer is valid.

Theorem 2 (A sufficient condition for valid transfers). *Let PN^N and PN^O be sound WF-nets, $PN^{ON} = PN^O \cup PN^N$, and let DC be the dynamic change region. For any reachable marking M of PN^O not marking the dynamic change region, i.e., $i_{PN^O} \xrightarrow{*} M$ in PN^O and $M(p) = 0$ for any $p \in DC \cap P^O$, a transfer $(PN^O, M) \Rightarrow (PN^N, M)$ is valid.*

Proof. See Appendix B □

Consider for example Figure 6. Theorem 2 guarantees that a case marking place $s2$ can be transferred from PN^O to PN^N and vice versa. Note that, just like the static change region, the dynamic change region is symmetric. The result of the algorithm does not depend on the role of PN^O and PN^N: The two roles can be reversed without changing the outcome of the algorithm. Also consider the other examples shown in figures 4, 5, 7, 8, and 9. If the dynamic change region indicated in either PN^O or PN^N is unmarked, a valid transfer is possible.

The examples given also indicate that Theorem 2 provides a sufficient but not necessary condition. Consider for example Figure 6. The dynamic change region includes $s3$, $s4$, $s5$, and $s6$. However, a transfer from any of these places is valid. The markings with in single token in $s3$, $s4$, $s5$, or $s6$ are reachable from $s1$ in both PN^O and PN^N. The following theorem gives a weaker condition for valid transfers. This theorem is based on the observation that only the internal places inside the dynamic change region may endanger the validity of the transfer. Places on the border of the dynamic change region, i.e., places connected to transitions outside DC, can be marked without compromising the validity of the transfer.

Theorem 3 (A weaker condition for valid transfers). *Let PN^N and PN^O be sound WF-nets, $PN^{ON} = PN^O \cup PN^N$, and let DC be the dynamic change region. For any reachable marking M of PN^O not marking the internal places of the dynamic change region, i.e., $i_{PN^O} \xrightarrow{*} M$ in PN^O and $M(p) = 0$ for any $p \in \{x \in DC \cap P^O \mid (\bullet x) \cup (x\bullet) \subseteq DC\}$, a transfer $(PN^O, M) \Rightarrow (PN^N, M)$ is valid.*

Proof. See Appendix B. □

This theorem shows that we can strengthen the result stated in Theorem 2 quite easily. The set of places considered in Theorem 3 is called the *minimal change region*. The minimal change region MC is defined as follows: $MC = \{p \in DC \mid (\bullet x) \cup (x\bullet) \subseteq DC\}$. The minimal change region includes all nodes of the dynamic change region except the so-called border places. Note that the minimal change region may be smaller that the static change region. Consider for example Figure 4: $SC = \{s2, s3, b, c\}$ and $MC = \{s3, b\}$. The minimal change regions of the examples shown in figures 5, 6, 7, 8, and 9 are $MC = \{s3, s4, s5, s6, b, c, d, e, f, g\}$ (Figure 5: $s2$ and $s7$ are removed), $MC = \{c, f, j, k\}$ (Figure 6: all places are removed), $MC = \{s3, s6, b, c, e, f\}$ (Figure 7: $s2$ and $s4$ are removed), $MC = \{s3, s6, s7, s8, e, f\}$ (Figure 8: b and c are removed), and $MC = \{s2, s3, s4, b, c, e\}$ (Figure 8: a and d are removed) respectively.

In Section 1, Figure 1 was used to illustrate the dynamic change problem. We did not refer to this example in this section because the place identifiers used in both WF-nets are different. The places in both nets have been named different to avoid confusion while explaining the dynamic change problem. However, it is clear that $s1$ and $p1$ are in essence the same place because their interconnection structures are the same. The same holds for $s5$ and $p6$, $s2$ and $p2$, $s4$ and $p5$. For the remaining places the correspondence is less clear. Let us rename $p1$ to $s1$, $p2$ to $s2$, $p5$ to $s4$, and $p6$ to $s5$ and calculate SC, DC, MC. The static change region SC consists of the following nodes: *p3*, *p4*, *s3*, *prepare_shipment*, *send_goods*, *send_bill*, and *record_shipment*. The dynamic change region DC consists of *p3*, *p4*, *s2*, *s3*, *s4*, *prepare_shipment*, *send_goods*, *send_bill*, and *record_shipment*. The minimal change region MC consists of *p3*, *p4*, *s2*, *s3*, *s4*, *send_goods* and *send_bill*.

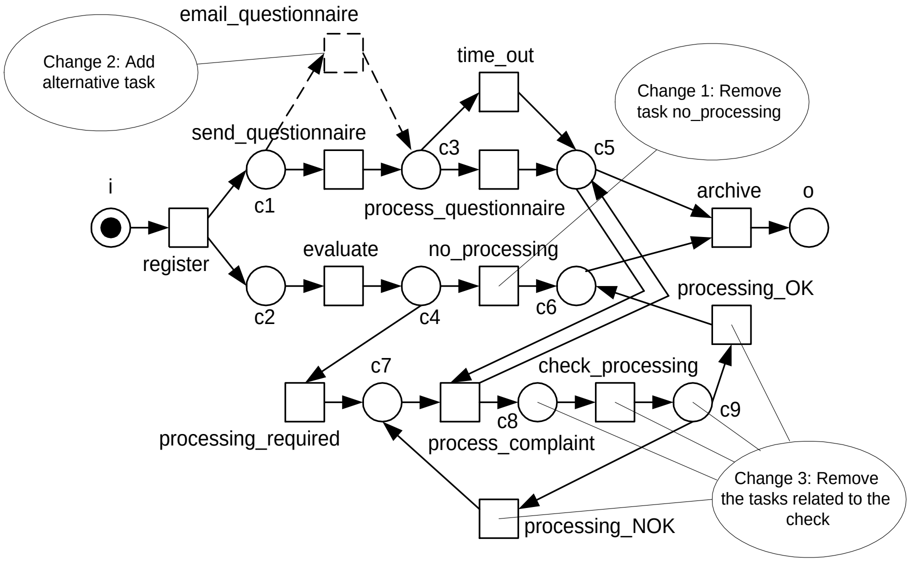

Fig. 10. Three potential changes.

Finally we illustrate the results using the complaint processing example introduced in Section 2.1. Figure 10 shows three potential changes. The first change corresponds to the removal of task *no_processing*. Assuming this change, the static change region SC consists of the following nodes: *c4*, *c6*, and *no_processing*. The dynamic change region DC coincides with the static change region. The minimal change region MC consists of only *no_processing*. Hence any transfer from the WF-net with task *no_processing* to the net without *no_processing* and vice versa is valid. The second change corresponds to the addition of an alternative task *email_questionnaire*. Assuming this change, the static change region SC consists of the following nodes: *c1*, *c3*, and *email_questionnaire*. The dynamic change region DC coincides with SC. The minimal change region MC consists of only the newly added task. Again any transfer is valid. The third change is less harmless. If *process_complaint* is connected directly to *c6* and the nodes *c8*, *c9*, *check_processing*, *processing_OK*, and *processing_NOK* are removed, then the resulting net is a sound WF-net. Assuming this change, the static change region SC consists of the following nodes: *c6*, *c7*, *c8*, *c9*, *process_complaint*, *check_processing*, *processing_OK*, and *processing_NOK*. The dynamic change region DC encompasses all nodes except *i* and *o*. The minimal change region MC consists of all nodes in-between *register* and *archive*. Hence only transfers from state *i* or *o* are guaranteed to be valid based on the minimal/dynamic change region.

5 Related work on dynamic change

There are many similarities between dynamic change in the workflow domain and *schema evolution* in the database domain. As the requirements of database applications change over time, the definition of the schema, i.e., the structure of the data elements stored in the database, is changed. Schema evolution has been an active field of research in the last decade (mainly in the field of object-oriented databases, cf. (Bertino and Martino, 1993)) and has resulted in techniques and tools that partially support the transformation of data from one database schema to another. Although dynamic change and schema evolution are similar, there are some additional complications in case of dynamic change. First, as was shown in the example of Figure 1, it is not always possible to transfer a case. Second, it is not acceptable to shut down the system, transfer all cases, and restart using the new procedure. Cases should be migrated while the system is running. Finally, dynamic change may introduce deadlocks and livelocks. The solutions provided by today's object-oriented databases do not deal with these complications. Therefore, we need new concepts and techniques.

Several researchers have worked on problems related to dynamic change. Ellis, Keddara, and Rozenberg (Ellis et al., 1995) propose a technique based on so-called "change regions." A change region contains all parts of a workflow process definition that potentially cause problems with respect to the transfer of cases. A change region has two versions; the old situation and the new situation. In this solution, there is one version of the complete process which covers the old and the new situation and changes affect cases as soon as possible. Parts of the workflow (i.e., change regions) become inactive after a while, because all old cases have been handled. This approach has the drawback that the process definition can become very complex (unless some automatic garbage collection is added). Another drawback is the fact that the authors do not provide a method for identifying the change region, i.e., change regions need to be identified manually. The authors do provide a notion of change correctness and give specific circumstances for which this is guaranteed. In (Ellis and Keddara, 2000a), the authors improve their approach by introducing jumpers. A jumper moves a case from the old workflow to the new workflow. The jump is postponed if for a state no jumper is available. Again, the authors do not give a concrete technique for the transfer of cases, i.e., jumpers are added manually. In (Ellis and Keddara, 2000b; Keddara, 1999), Keddara and Ellis present a language to support dynamic evolution within workflow systems (ML-DEWS). Based on the different modalities of change, the authors give a special purpose meta-language geared to model the workflow of change. Agostini and De Michelis (Agostini and Michelis, 2000) propose a technique for the automatic transfer of cases from an old process definition to a new process definition and also give criteria for determining whether a transfer is possible. The approach is interesting since it automatically computes the states for which it is not possible to migrate. Consider for example Figure 1. The approach presented in (Agostini and Michelis, 2000) indicates the necessity to postpone the transfer of running cases in state $[p_1, p_4]$. Unfortunately, the approach only works for a restricted class of workflows (e.g., the modeling language does not allow for iteration, although at runtime iteration can be achieved by backward jumps). A summary of this approach is given in (Michelis and Ellis, 1998). Weske (Vossen and Weske, 1999; Weske, 2000) considers dynamic workflow change using a model similar to the model used by IBM's MQSeries. In this model there is no iteration and also alternatives are synchronized. As a result the control flow is similar to a subclass of Petri nets: the so-called acyclic marked graphs. By exploiting these restrictions, relatively simple criteria can be obtained to guarantee the proper migration of an instance from one schema to another (Weske, 2000). Joeris and Herzog use linked State Charts to address the problem of posteriori flexibility (Joeris and Herzog, 1998). Casati, Ceri, and Pernici (Casati et al., 1998) tackle the problem of dynamic change via a set of transformation rules and partition the state space into a part that is aborted, a part that is transferred, a part that is handled the old way, and parts which are handled by hybrid process definitions (similar to the approach using change regions). Reichert and Dadam (Reichert and Dadam, 1998) use a similar approach. However, semantical issues such as errors introduced by swapping tasks, skipping tasks, or multiple executions of a task are not considered. Voorhoeve and Van der Aalst (Voorhoeve and Aalst, 1996; Voorhoeve and Aalst, 1997) also propose a fixed set of transformation rules to support dynamic change. However, the rules are not given explicitly at the net level and semantical issues are not considered. Van der Aalst and Basten (Aalst and Basten, 2001) propose an approach based on inheritance. This approach uses a set of generic inheritance-preserving transformation and transfer rules. Semantical errors such as the swapping of tasks, the skipping of tasks, and the multiple execution of tasks can be avoided by choosing the appropriate inheritance notion, e.g., projection inheritance guarantees that tasks cannot be skipped by transferring a case from the superclass to the subclass. Unfortunately, the approach is not useful if the new workflow is not a super or subclass of the old workflow. The reader interested in workflow change and Petri nets is also referred to (Aalst et al., 2000b) which contains several papers of the authors mentioned above. We also refer to the PhD thesis of Keddara (Keddara, 1999) for a more complete overview of related work on dynamic change.

The strength of the approach presented in this paper is that it can be applied in the context of arbitrary changes. Note that we did not assume the absence of certain routing constructs (i.e., sequential, conditional, parallel, and iterative are included) or restrict change to specific types of changes. Another feature of the approach is that the change regions are determined based on the structure of the workflow model (i.e., syntax) rather than the dynamics (i.e., a state space exploration). This facilitates implementation and yields change regions which are tangible to end-users.

6 Conclusion

This paper provides a pragmatic approach to tackle the dynamic change bug. Based on the syntactic changes in the graphical workflow model, three types of change regions are calculated. The static change region incorporates the parts of the workflow model *directly* effected by the change. The dynamic change region extends the static change region to incorporate the parts of the workflow model *indirectly* effected by the change. The minimal change region reduces the dynamic change region by eliminating border nodes. The minimal change region is a subset of the dynamic change region. The main result of this paper is that cases (i.e., workflow instances) which leave the minimal change region unmarked can be transferred from the old workflow to the new workflow without creating problems such as deadlocks and livelocks: Successful termination is guaranteed.

In the future, we plan to implement the approach presented in this paper using a commercial workflow management system. First, we plan to extend the workflow management system COSA (Thiel/Ley/COSA Solutions) with a feature to calculate the minimal change region and to enact valid transfers. This extension of COSA is quite straightforward since COSA is based on Petri nets and provides an API to remove and create cases in any state in any workflow. Second, we plan to realize the same functionality using other workflow management systems. Staffware (Staffware plc) is an example of another system we use in our laboratory. Implementation of this feature in Staffware is less straightforward because Staffware is not based on Petri nets and it is not known whether the required API is provided. Other candidates for realizing our approach are Verve (Verve Inc.) and i-Flow (Fujitsu Software Corporation). Both systems offer extensive API's.

Acknowledgements Part of the work reported in this paper was conducted while the author was visiting the Large Scale Distributed Information Systems (LSDIS) Laboratory, Department of Computer Science, University of Georgia, Athens, USA. The author would like to thank Ismael Budak Arpinar of the LSDIS lab. for his input during this visit, and Eric Verbeek for proofreading the paper.

References

Aalst, W. (1998a). Chapter 10: Three Good reasons for Using a Petri-net-based Workflow Management System. In Wakayama, T., Kannapan, S., Khoong, C., Navathe, S., and Yates, J., editors, *Information and Process Integration in Enterprises: Rethinking Documents*, volume 428 of *The Kluwer International Series in Engineering and Computer Science*, pages 161–182. Kluwer Academic Publishers, Boston, Massachusetts.

Aalst, W. (1998b). The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66.

Aalst, W. (2000). Workflow Verification: Finding Control-Flow Errors using Petri-net-based Techniques. In *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, Berlin.

Aalst, W. and Basten, T. (2001). Inheritance of Workflows: An approach to tackling problems related to change. *Theoretical Computer Science* (to appear).

Aalst, W., Basten, T., Verbeek, H., Verkoulen, P., and Voorhoeve, M. (2000a). Adaptive Workflow: On the Interplay between Flexibility and Support. In Filipe, J., editor, *Enterprise Information Systems*, pages 63–70. Kluwer Academic Publishers, Norwell.

Aalst, W., Desel, J., and Oberweis, A., editors (2000b). *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

Agostini, A. and Michelis, G. (2000). Improving Flexibility of Workflow Management Systems. In *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 218–234. Springer-Verlag, Berlin.

Bertino, E. and Martino, L. (1993). *Object-Oriented Database Systems: Concepts and Architecture*. Addison-Wesley.

Casati, F., Ceri, S., Pernici, B., and Pozzi, G. (1998). Workflow Evolution. *Data and Knowledge Engineering*, 24(3):211–238.

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

Ellis, C. and Keddara, K. (2000a). A Workflow Change Is a Workflow. In *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 201–217. Springer-Verlag, Berlin.

Ellis, C. and Keddara, K. (2000b). ML-DEWS: Modeling Language to Support Dynamic Evolution within Workflow Systems. *Computer Supported Cooperative Work*. To appear.

Ellis, C., Keddara, K., and Rozenberg, G. (1995). Dynamic change within workflow systems. In Comstock, N., Ellis, C., Kling, R., Mylopoulos, J., and Kaplan, S., editors, *Proceedings of the Conference on Organizational Computing Systems*, pages 10 – 21, Milpitas, California. ACM SIGOIS, ACM Press, New York.

Ellis, C. and Nutt, G. (1993). Modelling and Enactment of Workflow Systems. In Marsan, M. A., editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, Berlin.

Gostellow, K., Cerf, V., Estrin, G., and Volansky, S. (1972). Proper Termination of Flow-of-control in Programs Involving Concurrent Processes. *ACM Sigplan*, 7(11):15–27.

Jablonski, S. and Bussler, C. (1996). *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK.

Joeris, G. and Herzog, O. (1998). Managing evolving workflow specifications. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems, New York City, New York, USA, August 20-22*, pages 310–321. IEEE-CS Press.

Keddara, K. (1999). *Dynamic Evolution of Workflow Systems*. PhD thesis, University of Colorado, Boulder, Colorado, USA.

Kindler, E. and Aalst, W. (1999). Liveness, Fairness, and Recurrence. *Information Processing Letters*, 70(6):269–274.

Koulopoulos, T. (1995). *The Workflow Imperative*. Van Nostrand Reinhold, New York.

Lawrence, P., editor (1997). *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York.

Michelis, G. D. and Ellis, C. (1998). Computer Supported Cooperative Work and Petri Nets. In Reisig, W. and Rozenberg, G., editors, *Lectures on Petri Nets II: Applications*, volume 1492 of *Lecture Notes in Computer Science*, pages 125–153. Springer-Verlag, Berlin.

Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580.

Reichert, M. and Dadam, P. (1998). ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129.

Reisig, W. and Rozenberg, G., editors (1998). *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

Sadiq, S., Marjanovic, O., and Orlowska, M. (2000). Managing Change and Time in Dynamic Workflow Processes. *International Journal of Cooperative Information Systems*, 9(1-2):93–116.

Voorhoeve, M. and Aalst, W. (1996). Conservative Adaption of Workflow. In Wolf, M. and Reimer, U., editors, *Proceedings of the International Conference on Practical Aspects of Knowledge Management (PAKM'96), Workshop on Adaptive Workflow*, pages 1–15, Basel, Switzerland.

Voorhoeve, M. and Aalst, W. (1997). Ad-hoc Workflow: Problems and Solutions. In Wagner, R., editor, *Proceedings of the 8th DEXA International Workshop on Database and Expert Systems Applications*, pages 36–40, Toulouse, France. IEEE Computer Society Press, Los Alamitos, California, 1997.

Vossen, G. and Weske, M. (1999). The WASA2 Object-Oriented Workflow Management System. In Delis, A., Faloutsos, C., and Ghandeharizadeh, S., editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 587–589. ACM Press.

Weske, M. (2000). Foundation, Design, and Implementation of Dynamic Adaptations in a Workflow Management System. Fachbericht Angewandte Mathematik und Informatik 6/2000-I, Universität Münster, Münster, Germany.

WFMC (1996). Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011). Technical report, Workflow Management Coalition, Brussels.

A Petri nets

The classical Petri net (Desel and Esparza, 1995; Murata, 1989; Reisig and Rozenberg, 1998) is a directed bipartite graph with two node types called *places* and *transitions*. The nodes are connected via directed *arcs*. Connections between two nodes of the same type are not allowed. Places are represented by circles and transitions by rectangles.

Definition 6 (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)*

A place p is called an *input place* of a transition t iff there exists a directed arc from p to t. Place p is called an *output place* of transition t iff there exists a directed arc from t to p. We use $\bullet t$ to denote the set of input places for a transition t. The notations $t\bullet$, $\bullet p$ and $p\bullet$ have similar meanings, e.g., $p\bullet$ is the set of transitions sharing p as an input place. Note that we do not consider multiple arcs from one node to another. In the context of workflow procedures it makes no sense to have other weights, because places correspond to conditions.

To illustrate there concepts we consider the two Petri nets shown in Figure 1. The Petri net on the left has four transitions and five places. The Petri net on the right has four transitions and six places. Transition *prepare_shipment* in the left model has one input place and two output places. Note that *p1* and *s1* are source places, i.e., places without any input transition. Places *s5* and *p6* are sink places.

At any time a place contains zero or more *tokens*, drawn as black dots. The *state*, often referred to as marking, is the distribution of tokens over places, i.e., $M \in P \rightarrow \mathbb{N}$. We will represent a state as follows: $1p_1 + 2p_2 + 1p_3 + 0p_4$ is the state with one token in place p_1, two tokens in p_2, one token in p_3 and no tokens in p_4. We can also represent this state as follows: $p_1 + 2p_2 + p_3$. To compare states we define a partial ordering. For any two states M_1 and M_2, $M_1 \leq M_2$ iff for all $p \in P$: $M_1(p) \leq M_2(p)$

The number of tokens may change during the execution of the net. Transitions are the active components in a Petri net: they change the state of the net according to the following *firing rule*:

(1) A transition t is said to be *enabled* iff each input place p of t contains at least one token.

(2) An enabled transition may *fire*. If transition t fires, then t *consumes* one token from each input place p of t and *produces* one token for each output place p of t.

Given a Petri net (P, T, F) and a state M_1, we have the following notations:

- $M_1 \xrightarrow{t} M_2$: transition t is enabled in state M_1 and firing t in M_1 results in state M_2
- $M_1 \rightarrow M_2$: there is a transition t such that $M_1 \xrightarrow{t} M_2$
- $M_1 \xrightarrow{\sigma} M_n$: the firing sequence $\sigma = t_1 t_2 t_3 \ldots t_{n-1}$ leads from state M_1 to state M_n via a (possibly empty) set of intermediate states $M_2, ...M_{n-1}$, i.e., $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} ... \xrightarrow{t_{n-1}} M_n$

A state M_n is called *reachable* from M_1 (notation $M_1 \xrightarrow{*} M_n$) iff there is a firing sequence σ such that $M_1 \xrightarrow{\sigma} M_n$. Note that the empty firing sequence is also allowed, i.e., $M_1 \xrightarrow{*} M_1$.

We use (PN, M) to denote a Petri net PN with an initial state M. A state M' is a *reachable state* of (PN, M) iff $M \xrightarrow{*} M'$.

Consider the sequential Petri net shown in Figure 1 (i.e., the model on the right). If initially only place *s1* contains a token, only transition *prepare_shipment* is enabled. Firing this transition results in a state with just a token in *s2*, etc. Starting from the state with a token in *s1* five states are reachable. The parallel Petri net shown in Figure 1 has six reachable states when starting in the state with a token in *p1*. First transition *prepare_shipment* fires resulting in the state with a token in both *p2* and *p3*. From this state both *send_goods* and *send_bill* are enabled.

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

Definition 7 (Live). *A Petri net (PN, M) is live iff, for every reachable state M' and every transition t there is a state M'' reachable from M' which enables t.*

A Petri net is *structurally live* if there exists an initial state such that the net is live. None of the nets shown in Figure 1 is structurally live.

Definition 8 (Bounded, safe). *A Petri net (PN, M) is bounded iff for each place p there is a natural number n such that for every reachable state the number of tokens in p is less than n. The net is safe iff for each place the maximum number of tokens does not exceed 1.*

A Petri net is *structurally bounded* if the net is bounded for any initially state. Both nets shown in Figure 1 are structurally bounded.

Definition 9 (Well-formed). *A Petri net PN is well-formed iff there is a state M such that (PN, M) is live and bounded.*

Paths connect nodes by a sequence of arcs.

Definition 10 (Path, Elementary, Conflict-free). *Let PN be a Petri net. A path C from a node n_1 to a node n_k is a sequence $\langle n_1, n_2, \ldots, n_k \rangle$ such that $\langle n_i, n_{i+1} \rangle \in F$ for $1 \leq i \leq k-1$. C is elementary iff, for any two nodes n_i and n_j on C, $i \neq j \Rightarrow n_i \neq n_j$. C is conflict-free iff, for any place n_j on C and any transition n_i on C, $j \neq i-1 \Rightarrow n_j \notin \bullet n_i$.*

For convenience, we introduce the alphabet operator α on paths. If $C = \langle n_1, n_2, \ldots, n_k \rangle$, then $\alpha(C) = \{n_1, n_2, \ldots, n_k\}$. Moreover, we define $paths$ to be the function which returns the set of all elementary paths between two given nodes, i.e., $paths(a, b)$ is the set of elementary paths which start in node a and end in node b.

Definition 11 (Strongly connected). *A Petri net is strongly connected iff, for every pair of nodes (i.e., places and transitions) x and y, there is a path leading from x to y.*

Definition 12 (Free-choice). *A Petri net is a free-choice Petri net iff, for every two transitions t_1 and t_2, $\bullet t_1 \cap \bullet t_2 \neq \emptyset$ implies $\bullet t_1 = \bullet t_2$.*

Definition 13 (State machine). *A Petri net is state machine iff each transition has exactly one input and one output place.*

Definition 14 (S-component). *A subnet $PN_s = (P_s, T_s, F_s)$ is called an S-component of a Petri net $PN = (P, T, F)$ if $P_s \subseteq P$, $T_s \subseteq T$, $F_s \subseteq F$, PN_s is strongly connected, PN_s is a state machine, and for every $q \in P_s$ and $t \in T$: $(q, t) \in F \Rightarrow (q, t) \in F_s$ and $(t, q) \in F \Rightarrow (t, q) \in F_s$.*

Definition 15 (S-coverable). *A Petri net is S-coverable iff for any node there exist an S-component which contains this node.*

See (Desel and Esparza, 1995; Reisig and Rozenberg, 1998) for a more elaborate introduction to these standard notions. In addition to these standard notions we also define the some operators on nets, i.e., the union of two nets, the subnet notion, and the projection of a net onto a set of places and transitions.

Definition 16 (Union, subnet). *Let $PN_1 = (P_1, T_1, F_1)$ and $PN_2 = (P_2, T_2, F_2)$ be two Petri nets. The union of PN_1 and PN_2 is a Petri net $PN_{PN_1 \cup PN_2} = PN_1 \cup PN_2$, where $P_{PN_1 \cup PN_2} = P_1 \cup P_2$, $T_{PN_1 \cup PN_2} = T_1 \cup T_2$, and $F_{PN_1 \cup PN_2} = F_1 \cup F_2$. PN_1 is a subnet of PN_2, denoted as $PN_1 \subseteq PN_2$, iff $P_1 \subseteq P_2$, $T_1 \subseteq T_2$, and $F_1 \subseteq F_2$.*

Definition 17 (Projection). *Let $PN = (P, T, F)$ a Petri net and $X \subseteq P \cup T$. The projection of PN onto X is $PN|_X = (P \cap X, T \cap X, F \cap (X \times X))$.*

The projection of a Petri net onto a set of nodes includes all connections between these nodes, i.e., if two nodes are connected in PN and are both included in X, then these nodes are also connected in $PN|_X$. Note that, by definition, $PN|_X$ is a subnet of PN.

B Proof of theorems 2 and 3

In this appendix we will show that the dynamic change region indeed provides a criterion which is sufficient to guarantee the validity of transfers. The essence of the proof uses the fact that each component X_i identified by the algorithm is similar to a sound WF-net, i.e., a connected set of nodes with one unique source node and one unique sink node whose composite behavior is comparable to a single transition. To prove the central theorem of this paper we need to introduce *components* and *source* and *sink* nodes.

Definition 18 (Source and sink nodes). *Let $PN = (P, T, F)$ be a Petri net and $X \subseteq P \cup T$. $source(PN, X)$ is the set of source nodes of X and is defined as follows: $source(PN, X) = \{x \in X \mid \bullet x \cap X = \emptyset\}$. $sink(PN, X)$ is the set of sink nodes of X and is defined as follows: $sink(PN, X) = \{x \in X \mid x \bullet \cap X = \emptyset\}$.*

A source node is either a node without any input nodes or a node with only external input nodes. Consider for example Figure 4. Place $s2$ is the only source node of SC in PN^O. A sink node is either a node without any output nodes or a node with only external output nodes. Consider for example Figure 9. Place $s2$ is the only sink node of SC in PN^N. Both $s2$ and $s4$ are source and sink nodes of SC in PN^O.

Based on the notions of source and sink nodes we define components.

Definition 19 (Component). *Let $PN = (P, T, F)$ be a Petri net and $X \subseteq P \cup T$. X is a component of PN if and only if:*

(i) $source(PN, X)$ is a singleton, i.e., there is an a such that $\{a\} = source(PN, X)$,
(ii) $sink(PN, X)$ is a singleton, i.e., there is a b such that $\{b\} = sink(PN, X)$,
(iii) for each $x \in X$: x is on a directed path from a to b,
(iv) for each elementary path C from a to b (i.e., $C \in paths(a, b)$): $\alpha(C) \subseteq X$.

Components are similar to WF-nets embedded in a larger Petri net. However, in contrast to WF-nets, the source/sink node can be a transition instead of a place.

In line 5 of the algorithm the set X is partitioned into subsets X_i. The goal of the algorithm is to extend X such that these subsets correspond to components. The following lemma lists eight properties of the X_i components constructed by the algorithm. These properties will be used in the proof of Theorem 2.

Lemma 1. *Let PN^N and PN^O be sound WF-nets, $PN^{ON} = PN^O \cup PN^N$, and let DC be the dynamic change region. Let DC be partitioned into X_1, X_2, ... ,X_n such that:*

(a) $X_i \cap X_j = \emptyset$ *for all* $1 \leq i < j \leq n$
(b) $DC = \bigcup_{1 \leq i \leq n} X_i$
(c) $PN^{ON}|_{X_i}$ *is connected for all* $1 \leq i \leq n$
(d) $(\bullet X_i) \cap X_j = \emptyset$ *and* $(X_i \bullet) \cap X_j = \emptyset$ *for all* $1 \leq i < j \leq n$

Such partitioning always exists and is unique. The partitioning has the following properties:

(e) For all $1 \leq i \leq n$*:* X_i *is a component of* PN^{ON}*.*
(f) For all $1 \leq i \leq n$*:* $X_i \cap (P^O \cup T^O)$ *is a component of* PN^O*.*
(g) For all $1 \leq i \leq n$*:* $X_i \cap (P^N \cup T^N)$ *is a component of* PN^N*.*
(h) For all $1 \leq i \leq n$*:* $source(PN^{ON}, X_i) = source(PN^O, X_i \cap (P^O \cup T^O)) = source(PN^N, X_i \cap (P^N \cup T^N))$ *and* $sink(PN^{ON}, X_i) = sink(PN^O, X_i \cap (P^O \cup T^O)) = sink(PN^N, X_i \cap (P^N \cup T^N))$*.*

Proof. First we prove that DC can be partitioned in into X_1, X_2, ... ,X_n and that this partitioning is unique. Partition DC into singletons X_1, X_2, ... ,X_m. Such a partitioning satisfies properties (a), (b), and (c). If the partitioning does not satisfy property (d), then there is an i and j such that there is an arc from a node in X_i to a node in X_j. If this is the case, then join X_i and X_j. Clearly the nodes $X_i \cup X_j$ are connected. Then repeat the procedure until (d) holds. Note that the existence of such a partition is used in line 5 of the algorithm.

It remains to be proven that properties (e), (f), (g), and (h) hold. We will prove these properties for a given set of nodes X_i ($1 \leq i \leq n$) identified in the partitioning.

The algorithm stops if no new nodes are added in lines 6 through 14. This implies that at the end:

(i) For all $a \in X_i, b \in X_i \backslash \{a\}, c \in P^{ON} \cup T^{ON}, C_1 \in paths(a, c), C_2 \in paths(b, c)$ such that $\alpha(C_1) \cap \alpha(C_2) = \{c\}$: $\alpha(C_1) \cup \alpha(C_2) \subseteq X_i$.
(ii) For all $a \in X_i, b \in X_i \backslash \{a\}, c \in P^{ON} \cup T^{ON}, C_1 \in paths(c, a), C_2 \in paths(c, b)$ such that $\alpha(C_1) \cap \alpha(C_2) = \{c\}$: $\alpha(C_1) \cup \alpha(C_2) \subseteq X_i$.
(iii) For all $x \in X_i$ such that $\bullet x \cap X_i \neq \emptyset$: $\bullet x \subseteq X_i$, i.e., $\bullet x \not\subseteq X_i$ implies $x \in source(PN^{ON}, X_i)$.
(iv) For all $x \in X_i$ such that $x \bullet \cap X_i \neq \emptyset$: $x \bullet \subseteq X_i$, i.e., $x \bullet \not\subseteq X_i$ implies $x \in sink(PN^{ON}, X_i)$.

The first two observations follow directly from the algorithm. The latter two are the result of line 14 in the algorithm and property (d). These observations are used to prove the remaining properties.

Property (e). First we prove that $source(PN^{ON}, X_i)$ is a singleton. There is at least one source node. If X_i contains $i_{PN^{ON}}$, then $i_{PN^{ON}}$ is a source node. If X_i does not contain $i_{PN^{ON}}$, then there is a directed path from $i_{PN^{ON}}$ to a node in X_i. Consider the first node of X_i on this path. Clearly this node is a source node of X_i (use Property (iii)). There cannot be two source nodes. Suppose that both a and b are source nodes of X_i and $a \neq b$. There is a directed elementary path from $i_{PN^{ON}}$ to a and from $i_{PN^{ON}}$ to b. Let c be the last common node of these two paths, i.e., walk backwards on the directed elementary path from $i_{PN^{ON}}$ to a until one encounters a node also appearing in the other path. Based on these two paths and the last common node, we define two subpaths: an elementary directed path C_1 from c to a, and an elementary directed path C_2 from c to b. Clearly, $(\alpha(C_1) \cap \alpha(C_2)) = \{c\}$. Hence, $(\alpha(C_1) \cup \alpha(C_2)) \subseteq X_i$ (use Observation (i) and $\{c\} \subseteq X_i$). However, since both a and b are source nodes of X_i, there cannot be any input nodes from within X_i. Hence, both paths cannot contain multiple nodes, i.e., $c = a$ and $c = b$. This contradicts the assumption that $a \neq b$ and shows that there can only be one source node.

Similarly, it can be shown that $sink(PN^{ON}, X_i)$ is a singleton.

Let $\{a\} = source(PN^{ON}, X_i)$, $\{b\} = sink(PN^{ON}, X_i)$, and $x \in X_i$. We need to prove that x is on a directed path from a to b. Let C_1 be a directed path from $i_{PN^{ON}}$ to x and C_2 be a directed path from x to $o_{PN^{ON}}$. Such paths exist since PN^{ON} is a WF-net. Let y be the first element of X_i on C_1. Clearly $y = a$ (use Observation (iii)). Hence, there is a directed path from a to x. Similarly, it can be shown that there is a directed path from x to b.

Let C be an elementary path from a to b. Observation (i) implies that $\alpha(C) \subseteq X_i$ ($b = c$).

Property (f). Let a be the unique source node of X_i in PN^{ON}, i.e., $\{a\} = source$ (PN^{ON}, X_i). a is also a node of PN^O: either $a = i_{PN^{ON}}$ which also appears in PN^O or there is a node x not in DC such that $x \in \bullet a$. In the latter case x is not in the static change region ($SC \subseteq DC$) and therefore the set of nodes connected to x did not change. Hence a is a node of PN^O. Since PN^O is a subnet of PN^{ON}, a is also a source node of X_i in PN^O. Note that only by adding new connections source nodes can become non-source nodes. Similarly, we can show that the unique sink node b of X_i in PN^{ON}, i.e., $\{b\} = sink(PN^{ON}, X_i)$, is also a sink node of PN^O.

a is a source node of X_i in PN^O and b is a sink node of X_i in PN^O. Before we show that these two nodes are unique, we focus on the other two properties a component needs to satisfy.

Every node x in X_i is on a path from a to b in PN^{ON} (see proof of Property (e)). If $x \in X_i$ is a node of PN^O, then x is on a path from a to b in PN^O. Let C_1 be a directed path from i_{PN^O} to x and C_2 be a directed path from x to o_{PN^O} in PN^O. Such paths exist since PN^O is a WF-net and both paths are also paths of PN^{ON}. Is is easy to show that a must appear on path C_1 (consider the first node of X_i; this must be a) and b must appear on C_2.

Let C be a directed path from a to b in PN^O. C is also a path in PN^{ON}. Clearly, $\alpha(C) \subseteq X_i$ (see proof of Property (e)). Since C be a directed path in PN^O, $\alpha(C) \subseteq X_i \cap (P^O \cup T^O)$.

It remains to be proven that a and b are unique. Suppose there is an additional source node x, i.e., $x \in source(PN^O, X_i \cap (P^O \cup T^O))$ and $x \neq a$. Since x is on a directed path from a to b contained in $X_i \cap (P^O \cup T^O)$, x cannot be source node, i.e., a is the only source node. Similarly, it can be shown that b is unique.

Based on these observations we conclude that $X_i \cap (P^O \cup T^O)$ is a component of PN^O.

Property (g). The proof of this property is identical to the proof of Property (f).

Property (h). This property follows directly from the proof of properties (f) and (g). □

Based on the eight properties listed in Lemma 1, we prove Theorem 2. This theorem states that the dynamic change region provides a sufficient condition for valid transfers.

Theorem (A sufficient condition for valid transfers). *Let PN^N and PN^O be sound WF-nets, $PN^{ON} = PN^O \cup PN^N$, and let DC be the dynamic change region. For any reachable marking M of PN^O not marking the dynamic change region, i.e., $i_{PN^O} \xrightarrow{*} M$ in PN^O and $M(p) = 0$ for any $p \in DC \cap P^O$, a transfer $(PN^O, M) \Rightarrow (PN^N, M)$ is valid.*

Proof. Let M be such that $i_{PN^O} \xrightarrow{*} M$ in PN^O and $M(p) = 0$ for any $p \in DC \cap P^O$.

First, we prove that for all $p \in P^O : M(p) \geq 1$ implies $p \in P^N$. If $p \in P^O$ and $M(p) \geq 1$, then $p \notin DC$. Hence, $p \notin SC$ ($SC \subseteq DC$). Property 1 shows that $(P^O \cup T^O) \setminus (P^N \cup T^N) \subseteq SC$. Therefore, $p \in P^N$.

Finally, we prove that $i_{PN^N} \xrightarrow{*} M$ in PN^N. Lemma 1 shows that DC can be partitioned into X_1, X_2, ... ,X_n such that X_i is a component of PN^{ON}, $X_i \cap (P^O \cup T^O)$ is a component of PN^O, and $X_i \cap (P^N \cup T^N)$ is a component of PN^N. Consider an arbitrary component X_i with $\{a\} = source(PN^{ON}, X_i)$ and $\{b\} = sink(PN^{ON}, X_i)$. If both a and b are places, then $PN^{ON}|_{X_i}$, $PN^O|_{X_i}$, and $PN^N|_{X_i}$ are WF-nets. This follows directly from Definition 19. Since $PN^{ON}|_{X_i}$, $PN^O|_{X_i}$, and $PN^N|_{X_i}$ are subnets of sound WF-nets, these subnets are also sound. (See Theorem 3 in (Aalst, 2000).) Note that the soundness of each subnet heavily depends on the safeness of the enclosing WF-net. Since the subnets are sound their behavior corresponds to a single transition t^v connecting a and b. Now consider firing sequence σ which leads to M in PN^O, i.e., $i_{PN^O} \xrightarrow{\sigma} M$. Consider the transitions of X_i which occur in σ. These transition form complete subsequences of the embedded WF-net $PN^O|_{X_i}$, i.e., since no tokens are left in X_i every subsequence corresponds one firing of the virtual transition t^v. Each firing of this virtual transition can be mimicked by a firing sequence of the embedded WF-net $PN^N|_{X_i}$ in PN^N. This way occurrences of transitions in $X_i \cap T^O$ can be replaced by transitions in $X_i \cap T^N$. This assumes that both a and b are places. However, the same reasoning can be applied to components where a and/or b are transitions. Such a transition-bordered WF-net can be transformed into a sound WF-net by adding a source and/or sink place. This can be repeated for each of the components. Therefore, σ can be transformed into a firing sequence σ' which leads to M in PN^N, i.e., $i_{PN^O} \xrightarrow{\sigma'} M$. Hence, $i_{PN^N} \xrightarrow{*} M$ in PN^N. □

Finally we prove Theorem 3.

Theorem (A weaker condition for valid transfers). *Let PN^N and PN^O be sound WF-nets, $PN^{ON} = PN^O \cup PN^N$, and let DC be the dynamic change region. For any reachable marking M of PN^O not marking the internal places of the dynamic change region, i.e., $i_{PN^O} \xrightarrow{*} M$ in PN^O and $M(p) = 0$ for any $p \in \{x \in DC \cap P^O \mid (\bullet x) \cup (x\bullet) \subseteq DC\}$, a transfer $(PN^O, M) \Rightarrow (PN^N, M)$ is valid.*

Proof. Compared to Theorem 2 so-called border places p can be marked while a case is being transferred. Consider a place $p \in DC \cap P^O$ such that not $(\bullet p) \cup (p\bullet) \subseteq DC$, i.e., $\bullet p \not\subseteq DC$ or $p\bullet \not\subseteq DC$. If $\bullet p \not\subseteq DC$, then $\{p\} = source(PN^{ON}, X_i) = source(PN^O, X_i) = source(PN^N, X_i)$ of some component X_i. Since p appears in PN^O and PN^N, the sets of input transitions of p are identical in PN^O and PN^N, and $PN^O|_{X_i}$ and $PN^N|_{X_i}$ have a behavior similar to a single transition, a transfer of a token in p does not jeopardize the validity of the transfer. Similarly, a token in a place p with $p\bullet \not\subseteq DC$ cannot jeopardize the validity. Hence, if only places outside DC and border places are marked, then $(PN^O, M) \Rightarrow (PN^N, M)$ is valid. □

Biography Wil van der Aalst is a full professor of Information Systems and head of the Department of Information and Technology of the Faculty of Technology Management of Eindhoven University of Technology. He is also a part-time professor at the Computing Science department of the same university and has been working as a part-time consultant for Bakkenist. Wil van der Aalst's research interests include information systems, simulation, Petri nets, process models, workflow management systems, verification techniques, enterprise resource planning systems, computer supported cooperative work, and interorganizational business processes.