Inheritance of Workflows

An approach to tackling problems related to change

W.M.P. van der Aalst[1,3,⋆] and T. Basten[2,3]

[1] Dept. of Technology Management, Eindhoven University of Technology, The Netherlands
w.m.p.v.d.aalst@tm.tue.nl

[2] Dept. of Electrical Engineering, Eindhoven University of Technology, The Netherlands
a.a.basten@tue.nl

[3] Dept. of Computing Science, Eindhoven University of Technology, The Netherlands

Abstract

Inheritance is one of the key issues of object-orientation. The inheritance mechanism allows for the definition of a subclass which inherits the features of a specific superclass. When adapting a workflow process definition to specific needs (ad-hoc change) or changing the structure of the workflow process as a result of reengineering efforts (evolutionary change), inheritance concepts are useful to check whether the new workflow process inherits some desirable properties of the old workflow process. Today's workflow management systems have problems dealing with both ad-hoc changes and evolutionary changes. As a result, a workflow management system is not used to support dynamically changing workflow processes or the workflow processes are supported in a rigid manner, i.e., changes are not allowed or handled outside of the workflow management system. In this paper, we propose inheritance-preserving transformation rules for workflow processes and show that these rules can be used to avoid problems such as the "dynamic-change bug." The dynamic-change bug refers to errors introduced by migrating a case (i.e., a process instance) from an old process definition to a new one. A transfer from an old process to a new process can lead to duplication of work, skipping of tasks, deadlocks, and livelocks. Restricting change to the inheritance-preserving transformation rules guarantees transfers without any of these problems. Moreover, the transformation rules can also be used to extract aggregate management information in case more than one version of a workflow process cannot be avoided.

Key words: Workflow management, Petri nets, inheritance, adaptive workflow, dynamic change, management information

Contents

⋆Part of this work was done while the author was at sabbatical leave at the Large Scale Distributed Information Systems (LSDIS) laboratory of the University of Georgia.

1 Introduction

Workflow-management technology aims at the automated support and coordination of business processes to reduce costs and flow times, and to increase quality of service and productivity [31, 43, 44]. A critical challenge for workflow management systems is their ability to respond effectively to process changes [42, 63]. 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 [5]. 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.

Adaptive workflow aims at providing process support similar to contemporary workflow systems, but in such a way that the workflow system is able to deal with process changes. Recent papers and workshops show that the problems related to workflow change are difficult to solve [3, 5, 8, 10, 20, 26, 27, 36, 37, 42, 52, 54, 60, 63]. 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 process changes: (1) *ad-hoc changes* and (2) *evolutionary changes*. Ad-hoc changes are handled on a case-by-case basis and affect only one case (i.e., process instance) or a selected group of cases. The change is the result of an error, a rare event, or special demands of the customer. Exceptions often result in ad-hoc changes. A typical example of an ad-hoc change is the need to skip a task in case of an emergency. A workflow process definition resulting from an ad-hoc change is called a *variant* of the workflow process. Ad-hoc change typically leads to many variants of a given workflow process running in parallel. Evolutionary change is of a structural nature: From a certain moment in time, the workflow changes for all new cases to arrive at the system. The change is the result of a new business strategy, reengineering efforts, or a permanent alteration of external conditions (e.g., a change of law). Evolutionary change is typically initiated by the management to improve efficiency or responsiveness or is forced by legislature or changing market demands. A workflow process definition resulting from an evolutionary change is called a *version* of the workflow process. New cases are handled according to the most recent version of a process. Existing cases (i.e., work-in-progress) may also be influenced by an evolutionary change. Sometimes it is acceptable to handle running cases the old way. However, in many situations, cases need to be transferred from the old version to the new version.

Both ad-hoc and evolutionary change inevitably lead to one of the following two situations: Either there are multiple variants and/or versions which are active at the same time or cases need to be migrated from one variant/version to another. Today's workflow management systems have problems dealing with both situations. We use the term *dynamic-change problem* (cf. [26]) to refer to the anomalies caused by transferring cases from one process to another. The term *management-information problem* is used to refer to the problem of providing an aggregate overview of the work-in-progress in case of multiple versions and/or variants. The trend is towards an increasingly dynamic situation where both ad-hoc and evolutionary changes are needed to improve customer service and reduce costs continuously. Therefore, these problems are relevant for the next generation of workflow management systems.

In this paper, we use Petri nets to illustrate process-related concepts. In fact, we mainly use a restricted class of Petri nets, namely the class of so-called WF-nets [1, 2]. In a WF-net, there is one source place and one sink place and all other nodes are on a path from source to sink. Readers not familiar with Petri nets and workflow modeling are referred to Section 2.

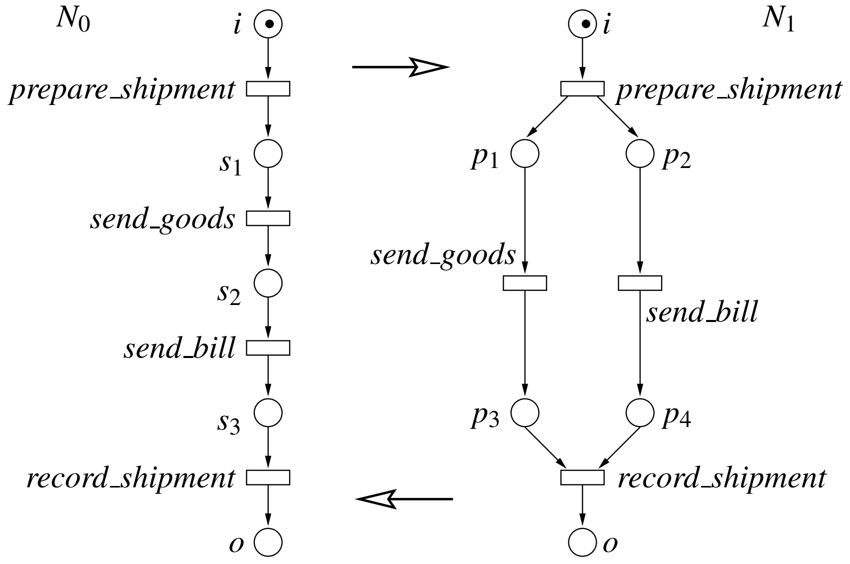

Figure 1.1: The dynamic-change bug.

Figure 1.1 shows two workflow process definitions illustrating the dynamic-change problem. 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 s_2 is mapped onto the state with a token in p_2 and p_3. 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, which means that cases need to be 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 state with a token in p_1 and a token in p_2 is mapped onto one token in s_1, and a state with a token in p_2 and a token in p_3 is mapped onto one token in s_2. However, the state with a token in both p_1 and p_4 (i.e., *prepare_shipment* and *send_bill* have been executed) causes problems because there is no corresponding state in the sequential process (where it is not possible to execute *send_bill* before *send_goods*). The example in Figure 1.1 shows that it is not straightforward to migrate old cases to the new process after a change.

The problem illustrated in Figure 1.1 is a result of reducing the degree of parallelism by making the process sequential. Similar problems occur when the ordering 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 a 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 workflow management systems truly supporting adaptive workflow. Today's workflow management systems are not able to handle this problem. These systems typically use a *versioning mechanism*, i.e., every change leads to a new version and cases refer 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 such a mechanism is not appropriate. If a case has a long flow time, then it is often not acceptable to handle existing cases the old way. Consider for example a process for handling mortgage loans. Mortgages typically have a duration of 20 to 30 years. If the mortgage process changes several times per year, this could lead to dozens of different versions running in parallel. To reduce costs 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 simultaneously. In fact, there may be legal reasons (i.e., starting from 1-1-2000 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 1.1 make a direct transfer hazardous. Note that the dynamic-change problem is relevant for both ad-hoc change and evolutionary change. However, the problem is most prominent for evolutionary change where potentially many cases need to be transferred.

Another problem related to change is the problem that it may lead to multiple *active* versions/variants of the same process which makes it difficult to provide *aggregate management information*. Consider again Figure 1.1. Assume that the two workflow process definitions are versions of the same workflow process. At some point in time, the left-hand process may contain six running cases, two in state s_1, three in state s_2, and one in state s_3, whereas the right-hand process may contain four running cases, two in the state with tokens in p_1 and p_2 and two in the state with tokens in p_1 and p_4. To provide aggregate management information, these numbers must be combined in such a way that the result provides a meaningful representation of the amount of work-in-progress. In the example, the solution is not very difficult because each state in the left-hand process definition of Figure 1.1 has a corresponding state in the right-hand process definition. As a result, aggregate management information can be collected by projecting the states of all cases onto the right-hand process definition. Doing so yields that, for four of the total of ten cases, tasks *send_goods*, *send_bill*, and *record_shipment* still need to be executed; for two cases, *send_goods* and *record_shipment* still need to be performed; three cases are in a state that *send_bill* and *record_shipment* still need to be done, whereas for one case only task *record_shipment* still needs to be performed. It is possible to summarize this information by counting the number of tokens resulting in each place of the right-hand process definition of Figure 1.1 when projecting the ten cases onto this process definition: Places i and o do not contain any tokens, place p_1 contains six tokens, place p_2 contains seven tokens, place p_3 contains four tokens, and, finally, p_4 contains three tokens. Although this example is not very complicated, in general, it is not straightforward to obtain aggregate management information when the different process definitions are more complex or their number is larger.

The management-information problem explained above occurs if multiple versions and/or variants of the workflow process cannot be avoided. For evolutionary change, the number of versions is often limited. In fact, if all cases are transferred, then there is just one active version (i.e., all running cases use the same version). However, in some situations, it is not possible nor desirable to transfer cases to the most recent process. There can be legal, managerial, or practical reasons that prevent the transfer of cases. In such a situation, there are multiple active versions of the same process. Ad-hoc change may lead to the situation where the number of variants may be of the same order of magnitude as the number of cases. The variants are customized to accommodate specific needs. To manage a workflow process with different versions/variants, it is desirable to have an aggregated view of the work-in-progress. Note that in a manufacturing process the manager can get a good impression of the work-in-progress by walking through the factory. For a workflow process handling digitized information, this is not possible. Therefore, it is of the utmost importance to supply the manager with tools to obtain a condensed but accurate view of the workflow processes. Although the problem of extracting aggregate management information is relevant for both ad-hoc and evolutionary change, it is most prominent for ad-hoc change.

To tackle the dynamic-change problem and the management-information problem, we propose an approach based on the *inheritance-preserving transformation rules* introduced in [15, 14, 4, 16]. Inheritance is one of the key concepts of object-orientation. Classes and objects in object-oriented design correspond to workflow process definitions and cases in a workflow management context. In object-oriented design, inheritance is typically restricted to the static aspects (e.g., data and methods) of an object class. For workflow management, the dynamic behavior of cases is of prime importance. The inheritance-preserving transformation rules used in this paper focus on workflow process definitions in a Petri-net-based setting. The four inheritance relations presented in this paper use branching bisimilarity (to compare processes) in combination with the notions of *encapsulation* and *abstraction*. Encapsulation corresponds to blocking tasks, whereas abstraction corresponds to hiding tasks.[1] Restricting process changes to the inheritance-preserving transformation rules presented in this paper makes a direct transfer possible in any state while avoiding problems such as the one illustrated by Figure 1.1. Note that the inheritance rules can only be used to *avoid* the dynamic-change bug, i.e., it is a preventive treatment of the problem. If changes such as the one shown in Figure 1.1 are allowed, the only cure is to postpone the transfer in case of problems. As a result, in such a case, there may be several active versions of the same workflow process. There may be other reasons for having multiple active versions, e.g., by law, cases are forced to be handled the old way. In case of ad-hoc workflow, there are also multiple active versions of the same process (called variants). The presence of multiple active versions and/or variants of the same process can obscure the status of the whole workflow.

[1] The notions of encapsulation and abstraction in this paper are inspired by *process-algebraic concepts* (see [12]). In process algebra, the terms "encapsulation" and "abstraction" have a different meaning than the same terms in object-oriented design.

Fortunately, the inheritance-preserving transformation rules can also be used to construct aggregate management information. The inheritance notions allow for the definition of concepts such as a Maximal/Greatest Common Divisor (MCD/GCD) and Minimal/Least Common Multiple (MCM/LCM) of a set of variants/versions. These concepts can be used to create a condensed overview of the work-in-progress. Clearly, the dynamic-change problem and the management-information problem are related. By solving the dynamic-change problem (i.e., instantly migrating all cases to a single version of the process), there is no need to construct aggregate management information because there is just one active version. However, ad-hoc changes inevitably lead to multiple variants and, as illustrated by Figure 1.1, multiple active versions of a workflow process are sometimes unavoidable.

The remainder of this paper is organized as follows. In Section 2, 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 [1, 2]. Section 3 introduces the inheritance notions and the inheritance-preserving transformation rules used in this paper. In Section 4, the use of inheritance in a workflow-management context is discussed. Section 5 tackles the problems related to dynamic change using the inheritance-preserving transformation rules. In Section 6, it is shown that the results can also be used to create aggregate management information. In Section 7, we consider the use of tools to support the notions presented in this paper. Finally, Section 8 summarizes the results.

2 Preliminaries

This section introduces the techniques used in the remainder. Standard definitions for bags and Petri nets are given. Moreover, more advanced concepts such as branching bisimilarity, workflow nets, and soundness are presented. These preliminaries are required to define the inheritance concepts in an unambiguous way.

2.1 Notations for bags

In this paper, bags are defined as finite multi-sets of elements from some alphabet A. A bag over alphabet A can be considered as a function from A to the natural numbers $\mathbb{N}$ such that only a finite number of elements from A is assigned a non-zero function value. For some bag X over alphabet A and $a \in A$, $X(a)$ denotes the number of occurrences of a in X, often called the cardinality of a in X. The set of all bags over A is denoted $\mathcal{B}(A)$. For the explicit enumeration of a bag, a notation similar to the notation for sets is used, but using square brackets instead of curly brackets and using superscripts to denote the cardinality of the elements. For example, $[a^2, b, c^3]$ denotes the bag with two elements a, one b, and three elements c; the bag $[a^2 \mid P(a)]$ contains two elements a for every a such that $P(a)$ holds, where P is some predicate on symbols of the alphabet under consideration. To denote individual elements of a bag, the same symbol "$\in$" is used as for sets: For any bag X over alphabet A and element $a \in A$, $a \in X$ if and only if $X(a) > 0$. The sum of two bags X and Y, denoted $X + Y$, is defined as $[a^n \mid a \in A \wedge n = X(a) + Y(a)]$. The difference of X and Y, denoted $X - Y$, is defined as $[a^n \mid a \in A \wedge n = (X(a) - Y(a)) \max 0]$. The binding of sum and difference is left-associative. The restriction of X to some domain $D \subseteq A$, denoted $X \lceil D$, is defined as $[a^{X(a)} \mid a \in D]$. Restriction binds stronger than sum and difference. The notion of subbags is defined as expected: Bag X is a subbag of Y, denoted $X \leq Y$, if and only if, for all $a \in A$, $X(a) \leq Y(a)$. Note that any finite set of elements from A also denotes a unique bag over A, namely the function yielding 1 for every element in the set and 0 otherwise. Therefore, finite sets can also be used as bags. If X is a bag over A and Y

is a finite subset of A, then $X - Y$, $X + Y$, $Y - X$, and $Y + X$ yield bags over A. Moreover, $X \leq Y$ and $Y \leq X$ are defined in a straightforward manner.

2.2 Labeled Place/Transition nets

In this section, we define a variant of the classic Petri-net model, namely labeled Place/Transition nets. For a more elaborate introduction to Petri nets, the reader is referred to [24, 46, 53]. Let U be some universe of identifiers; let L be some set of *action labels*.

Definition 2.1. (Labeled P/T-net) An L-labeled Place/Transition net, or simply labeled P/T-net, is a tuple (P, T, F, ℓ) where:

1. $P \subseteq U$ is a finite set of *places*,
2. $T \subseteq U$ is a finite set of *transitions* such that $P \cap T = \emptyset$,
3. $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, called the *flow relation*, and
4. $\ell : T \to L$ is a *labeling function*.

In the Petri-net literature, the class of Petri nets introduced in Definition 2.1 is sometimes referred to as the class of (labeled) *ordinary* P/T-nets to distinguish it from the class of Petri nets that allows more than one arc between a place and a transition.

Let (P, T, F, ℓ) be a labeled P/T-net. Elements of $P \cup T$ are referred to as *nodes*. A node $x \in P \cup T$ is called an *input node* of another node $y \in P \cup T$ if and only if there exists a directed arc from x to y; that is, if and only if xFy. Node x is called an *output node* of y if and only if there exists a directed arc from y to x. If x is a place in P, it is called an input place or an output place; if it is a transition, it is called an input or an output transition. The set of all input nodes of some node x is called the *preset* of x; its set of output nodes is called the *postset*. Two auxiliary functions $\bullet\_, \_\bullet : (P \cup T) \to \mathcal{P}(P \cup T)$ are defined that assign to each node its preset and postset, respectively. For any node $x \in P \cup T$, $\bullet x = \{y \mid yFx\}$ and $x\bullet = \{y \mid xFy\}$. Note that the preset and postset functions depend on the context, i.e., the P/T-net the function applies to. If a node is used in several nets, it is not always clear to which P/T-net the preset/postset functions refer. Therefore, we augment the preset and postset notation with the name of the net whenever confusion is possible: $\overset{N}{\bullet}x$ is the preset of node x in net N and $x\overset{N}{\bullet}$ is the postset of node x in net N.

A labeled P/T-net as defined above is a static structure. Figure 2.2 shows the graphical representation of a P/T-net. Places are represented by circles; transitions are represented by rectangles. Attached to each place is its identifier. Attached to each transition is its label. Transition labeling is needed for two reasons. First, a P/T-net modeling a workflow process may contain several transitions referring to a single task (identified by the label) in the workflow process. Second, we use transition labels as a mechanism to abstract from tasks. For the sake of simplicity, we assume that transition labels are identical to transition identifiers unless explicitly stated otherwise.

Labeled P/T-nets have a dynamic behavior. The behavior of a net is determined by its structure and its *state*. To express the state of a net, its places may contain *tokens*. In labeled P/T-nets, tokens are nothing more than simple markers (see Figure 2.2). The distribution of tokens over the places is often called the *marking* of the net.

Definition 2.3. (Marked, labeled P/T-net) A *marked*, L-labeled P/T-net is a pair (N, s), where $N = (P, T, F, \ell)$ is an L-labeled P/T-net and where s is a bag over P denoting the marking of the net. The set of all marked, L-labeled P/T-nets is denoted $\mathcal{N}$.

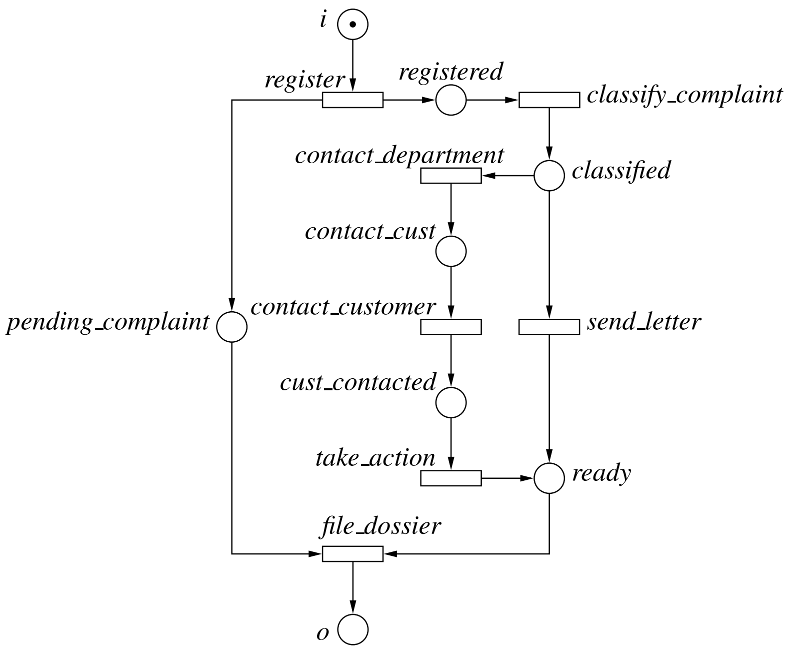

Figure 2.2: A labeled P/T-net.

The dynamic behavior of marked, labeled P/T-nets is defined by a so-called *firing rule*, which is simply a transition relation defining the change in the state of a marked net when executing an action. To define the firing rule, it is necessary to formalize when a net is allowed to execute a certain action.

Definition 2.4. (Transition enabling) Let (N, s) be a marked, labeled P/T-net in $\mathcal{N}$, where $N = (P, T, F, \ell)$. A transition $t \in T$ is *enabled*, denoted $(N, s)[t\rangle$, if and only if each of its input places p contains a token. That is, $(N, s)[t\rangle \Leftrightarrow \bullet t \leq s$.

When a transition t of a labeled P/T-net is enabled, the net can *fire* this transition. Upon firing, t removes a token from each of its input places and adds a token to each of its output places. This means that upon firing t, the marked net (N, s) changes into another marked net $(N, s - \bullet t + t\bullet)$.

Definition 2.5. (Firing rule) The firing rule $\_\,[\_\rangle\,\_ \subseteq \mathcal{N} \times L \times \mathcal{N}$ is the smallest relation satisfying for any (N, s) in $\mathcal{N}$, with $N = (P, T, F, \ell)$, and any $t \in T$,
$(N, s)[t\rangle \Rightarrow (N, s)\ [\ell(t)\rangle\ (N, s - \bullet t + t\bullet)$.

The labeled P/T-net shown in Figure 2.2 is used to illustrate the firing rule. The net models the processing of complaints by the complaints desk of a fictitious Company X. The complaints desk handles complaints of customers about the products produced by Company X. Each complaint is registered before it is classified. Depending on the classification of the complaint, a letter is sent to the customer or an inquiry is started. The inquiry starts with a consultation of the department involved, followed by a discussion with the customer. Based on this inquiry, the necessary actions are taken. Finally, the dossier is filed. Figure 2.2 shows the process definition which is used to configure the workflow management system used by the employees of the complaints desk. The marking shown in Figure 2.2 is $[i]$, i.e., the state with one token in place i. Transition *register* is the only transition enabled in this marking. Firing *register* results in the state [*pending_complaint*, *registered*], i.e., two tokens are produced. Then, *classify_complaint* will fire followed by either *send_letter* or *contact_department*, *contact_customer*, and *take_action*. Finally, *file_dossier* will fire. Note that *file_dossier* consumes two tokens and produces one token.

The firing rule determines the set of so-called *reachable* markings of a marked P/T-net. A marking s is reachable from the initial marking s_0 of a marked net (N, s_0) if and only if there exists a *sequence* of enabled transitions whose execution leads from s_0 to s. This paper uses the following notations for sequences. Let A be some alphabet of identifiers. A *sequence of length* n, for some natural number $n \in \mathbb{N}$, over alphabet A is a function $\sigma : \{0, \ldots, n-1\} \to A$. The sequence of length zero is called the empty sequence and written ε. For the sake of readability, a sequence of positive length is usually written by juxtaposing the function values: For example, a sequence $\sigma = \{(0, a), (1, a), (2, b)\}$, for $a, b \in A$, is written aab. The set of all sequences of arbitrary length over alphabet A is written A^*.

Definition 2.6. (Firing sequence) Let (N, s_0) with $N = (P, T, F, \ell)$ be a marked, labeled P/T-net in $\mathcal{N}$. A sequence $\sigma \in T^*$ is called a *firing sequence* of (N, s_0) if and only if, for some natural number $n \in \mathbb{N}$, there exist markings $s_1, \ldots, s_n \in \mathcal{B}(P)$ and transitions $t_1, \ldots, t_n \in T$ such that $\sigma = t_1 \ldots t_n$ and, for all i with $0 \leq i < n$, $(N, s_i)[t_{i+1}\rangle$ and $s_{i+1} = s_i - \bullet t_{i+1} + t_{i+1}\bullet$. (Note that $n = 0$ implies that $\sigma = \varepsilon$ and that ε is a firing sequence of (N, s_0).) Sequence σ is said to be *enabled* in marking s_0, denoted $(N, s_0)[\sigma\rangle$. Firing the sequence σ results in the unique marking s_n, denoted $(N, s_0)\ [\sigma\rangle\ (N, s_n)$.

The marked, labeled P/T-net $(N, [i])$ shown in Figure 2.2 has many enabled firing sequences. For example, firing sequence *register classify_complaint contact_department* is enabled. Executing this sequence results in marking [*pending_complaint*, *contact_cust*].

As mentioned, a marking of a labeled P/T-net is reachable if and only if there is a firing sequence leading from the initial marking to that marking.

Definition 2.7. (Reachable markings) The set of *reachable markings* of a marked, labeled P/T-net $(N, s) \in \mathcal{N}$ with $N = (P, T, F, \ell)$, denoted $[N, s\rangle$, is defined as the set $\{s' \in \mathcal{B}(P) \mid (\exists \sigma : \sigma \in T^* : (N, s)\ [\sigma\rangle\ (N, s'))\}$.

Consider for example the marked, labeled P/T-net $(N, [i])$ shown in Figure 2.2. There are two firing sequences leading to marking $[o]$. Therefore, $[o]$ is reachable. In total, there are seven markings reachable from $[i]$.

For the purpose of analyzing processes defined by P/T-nets, many properties have been defined and studied. Some properties refer to the net structure, while others refer to the dynamic behavior of a marked P/T-net. The following two definitions refer to structural properties. The first definition uses the standard notations for the inverse of a relation R (R^{-1}) and the reflexive and transitive closure of R (R^*).

Definition 2.8. (Connectedness) A labeled P/T-net $N = (P, T, F, \ell)$ is *weakly connected*, or simply *connected*, if and only if, for every two nodes x and y in $P \cup T$, $x(F \cup F^{-1})^* y$. Net N is *strongly connected* if and only if, for every two nodes x and y in $P \cup T$, xF^*y.

In the remainder of this paper, we assume all nets to be weakly connected. Moreover, we assume all nets to have at least two nodes. Nets without places or transitions do not make any sense.

Another structural property is the so-called *free-choice* property.

Definition 2.9. (Free-choice P/T-net) A *free-choice* P/T-net is a (labeled) P/T-net (P, T, F, ℓ) as in Definition 2.1 such that, for all transitions $t, u \in T$, either $\bullet t \cap \bullet u = \emptyset$ or $\bullet t = \bullet u$.

Free-choice P/T-nets are characterized by the fact that two transitions sharing an input place always share all their input places. From a pragmatic point of view, the class of free-choice P/T-nets is of particular interest; many workflow management systems use a diagramming technique which corresponds to free-choice nets. The class of free-choice P/T-nets combines a reasonable expressive power with strong analysis techniques. Consequently, free-choice P/T-nets have been extensively studied in the literature. The most important results on free-choice P/T-nets have been brought together in [24].

An example of a property which refers to the dynamics of a marked P/T net is boundedness.

Definition 2.10. (Boundedness) A marked, labeled P/T-net $(N, s) \in \mathcal{N}$ is *bounded* if and only if the set of reachable markings $[N, s\rangle$ is finite.

In a bounded net, the number of tokens in any place is bounded. If the maximum number of tokens in each place is one, then the net is safe.

Definition 2.11. (Safeness) A marked, labeled P/T-net $(N, s) \in \mathcal{N}$ with $N = (P, T, F, \ell)$ is *safe* if and only if, for any reachable marking $s' \in [N, s\rangle$ and any place $p \in P$, $s'(p) \leq 1$.

Note that safeness implies boundedness.

A transition is dead if and only if there is no reachable marking enabling that transition.

Definition 2.12. (Dead transition) Let (N, s) be a marked, labeled P/T-net in $\mathcal{N}$. A transition $t \in T$ is *dead* in (N, s) if and only if there is no reachable marking $s' \in [N, s\rangle$ such that $(N, s')[t\rangle$.

A property stronger than the absence of dead transitions is liveness. A P/T-net is live if and only if, no matter what marking has been reached, it is always possible to enable an *arbitrary* transition of the net by firing a number of other transitions.

Definition 2.13. (Liveness) A marked, labeled P/T-net $(N, s) \in \mathcal{N}$ with $N = (P, T, F, \ell)$ is *live* if and only if, for every reachable marking $s' \in [N, s\rangle$ and transition $t \in T$, there is a reachable marking $s'' \in [N, s'\rangle$ such that $(N, s'')[t\rangle$.

2.3 Branching bisimilarity

To formalize the inheritance concepts mentioned in the introduction, we need to formalize a notion of equivalence. Labeled P/T-nets are equipped with an equivalence relation that specifies when two different marked, labeled P/T-nets have the same (observable) behavior. By choosing different equivalence relations different semantics are obtained. For more information on the different semantics for concurrent systems the reader is referred to [32, 50]. In this paper, we use *branching bisimilarity* [34] as the standard equivalence relation on marked, labeled P/T-nets in $\mathcal{N}$.

The notion of a *silent action* is pivotal to the definition of branching bisimilarity. Silent actions are actions (i.e., transition firings) that cannot be observed. Silent actions are denoted with the label τ, i.e., only transitions in a P/T-net with a label different from τ are observable. Note that we assume that τ is an element of L. The τ-labeled transitions are used to distinguish between external, or observable, and internal, or silent, behavior, A single label is sufficient, since all internal actions are equal in the sense that they do not have any visible effects.

As explained in the next subsection, in the context of workflow management, we want to distinguish *successful termination* from *deadlock*. A *termination predicate* defines in what states a marked P/T-net can terminate successfully. If a marked, labeled P/T-net is in a state where it cannot perform any actions or terminate successfully, then it is said to be in a *deadlock*. Assume that $\downarrow \subseteq \mathcal{N}$ is some arbitrary termination predicate.

To define branching bisimilarity, two auxiliary definitions are needed: (1) a relation expressing that a marked, labeled P/T-net can evolve into another marked, labeled P/T-net by executing a sequence of zero or more τ actions; (2) a predicate expressing that a marked, labeled P/T-net can terminate by performing zero or more τ actions.

Definition 2.14. The relation $\_ \Longrightarrow \_ \subseteq \mathcal{N} \times \mathcal{N}$ is defined as the smallest relation satisfying, for any $p, p', p'' \in \mathcal{N}$, $p \Longrightarrow p$ and $(p \Longrightarrow p' \wedge p' \, [\tau\rangle \, p'') \;\Rightarrow\; p \Longrightarrow p''$.

Definition 2.15. The predicate $\Downarrow \_ \subseteq \mathcal{N}$ is defined as the smallest set of marked, labeled P/T-nets satisfying, for any $p, p' \in \mathcal{N}$, $\downarrow p \Rightarrow \Downarrow p$ and $(\Downarrow p \wedge p' \, [\tau\rangle \, p) \;\Rightarrow\; \Downarrow p'$.

Let, for any two marked, labeled P/T-nets $p, p' \in \mathcal{N}$ and action $\alpha \in L$, $p \, [(\alpha)\rangle \, p'$ be an abbreviation of the predicate $(\alpha = \tau \wedge p = p') \vee p \, [\alpha\rangle \, p'$. Thus, $p \, [(\tau)\rangle \, p'$ means that zero τ actions are performed, when the first disjunct of the predicate is satisfied, or that one τ action is performed, when the second disjunct is satisfied. For any observable action $a \in L \backslash \{\tau\}$, the first disjunct of the predicate can never be satisfied. Hence, $p \, [(a)\rangle \, p'$ is simply equal to $p \, [a\rangle \, p'$, meaning that a single a action is performed.

Definition 2.16. (Branching bisimilarity) A binary relation $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{N}$ is called a *branching bisimulation* if and only if, for any $p, p', q, q' \in \mathcal{N}$ and $\alpha \in L$,

1. $p\mathcal{R}q \wedge p \, [\alpha\rangle \, p' \Rightarrow$
   $(\exists \, q', q'' : \, q', q'' \in \mathcal{N} : \, q \Longrightarrow q'' \wedge q'' \, [(\alpha)\rangle \, q' \wedge p\mathcal{R}q'' \wedge p'\mathcal{R}q')$,
2. $p\mathcal{R}q \wedge q \, [\alpha\rangle \, q' \Rightarrow$
   $(\exists \, p', p'' : \, p', p'' \in \mathcal{N} : \, p \Longrightarrow p'' \wedge p'' \, [(\alpha)\rangle \, p' \wedge p''\mathcal{R}q \wedge p'\mathcal{R}q')$, and
3. $p\mathcal{R}q \Rightarrow (\downarrow p \Rightarrow \Downarrow q \wedge \downarrow q \Rightarrow \Downarrow p)$.

Two marked, labeled P/T-nets are called *branching bisimilar*, denoted $p \sim_b q$, if and only if there exists a branching bisimulation $\mathcal{R}$ such that $p\mathcal{R}q$.

$$\begin{tikzcd}
p \arrow[r, dash, dashed] \arrow[d, "\tau"'] \arrow[dr, dash, dashed] & q \arrow[d, Rightarrow] \\
p' \arrow[r, dash, dashed] & q'' = q'
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
p \arrow[r, dash, dashed] \arrow[d, "\alpha"'] \arrow[dr, dash, dashed] & q \arrow[d, Rightarrow] \\
p' \arrow[dr, dash, dashed] & q'' \arrow[d, "\alpha"] \\
 & q'
\end{tikzcd}$$

Figure 2.17: The essence of a branching bisimulation.

Figure 2.17 shows the essence of a branching bisimulation. The firing rule is depicted by arrows. The dashed lines represent a branching bisimulation. A marked, labeled P/T-net must be able to simulate any action of an equivalent marked net after performing any number of silent actions, except for a silent action which it may or may not simulate. The third property in Definition 2.16 guarantees that related marked nets always have the same termination options.

Branching bisimilarity is an equivalence relation on $\mathcal{N}$, i.e., $\sim_b$ is reflexive, symmetric, and transitive.

Property 2.18. *Branching bisimilarity, $\sim_b$, is an equivalence relation.*

Proof. See [14] for a detailed proof. □

Branching bisimilarity was first introduced in [33]. The definition given in this subsection is slightly different from the original definition. In fact, it is the definition of *semi*-branching bisimilarity, which was first defined in [62]. It can be shown that the two notions are equivalent in the sense that they define the same equivalence relation on marked, labeled P/T-nets [34, 13]. The reason for using the alternative definition is that it is more concise and more intuitive than the original definition. A comparison of the two definitions can be found in [13].

2.4 WF-nets

The use of Petri nets for workflow modeling has been suggested by many authors (e.g., [8, 9, 28, 40, 47]) and several workflow management systems use Petri nets as a design language, e.g., COSA [56], INCOME [51], and BaanWorkflow [11]. In fact, most commercial workflow management systems use a modeling language which corresponds to a subset of Petri nets (typically free-choice P/T-nets [2]).

Before we present the class of nets we use in the remainder of this paper, we introduce the basic concepts and terminology used in the workflow-management domain. These are the concepts supported by today's workflow management systems and also recognized by standardization bodies such as the Workflow Management Coalition [44].

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 complaint, 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 for a case and in what order (i.e., the life cycle of one case in isolation). Alternative terms for a 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 which needs to be executed for a specific case is called a *work item*. An example of a work item is the order to 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). In most offices, the resources are mainly human. However, because workflow management is not restricted to offices, we prefer the term resource. Resources are allowed to deal with specific work items. 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* (e.g., team, branch, or department). A work item which is being executed by a specific resource is called an *activity*. If we take a photograph of the state of a workflow, we see cases, work items, and activities. Work items link cases and tasks. Activities link cases, tasks, and resources.

In this paper, we abstract from the resources and focus on the process aspect. In fact, we only consider the *life cycle* of one case in isolation. Cases only interact with each other via competition for resources. The problems introduced in Section 1 are not related to the allocation of resources to tasks or the interaction between cases. Therefore, given the topic of this paper, it is reasonable to abstract from resources and to consider just one case at a time. We also abstract from *workflow attributes*. A workflow attribute is a specific piece of information used for the routing of a case. One can think of a workflow attribute as a control variable or a logistic parameter. A workflow attribute may be the age of a customer, the department responsible, or the registration date, and is used to make routing decisions. We abstract from these workflow attributes for the following reasons. In reality, the routing decisions (i.e., OR-splits) are based on workflow attributes whose values depend on application data and/or the behavior of the persons and applications involved. Since workflow attributes are typically set by external entities (i.e., resources, applications, or electronic messages), they cannot be modeled accurately. Therefore, we consider each choice to be a non-deterministic one. We also abstract from workflow attributes because it allows us to use P/T-nets rather than high-level Petri nets. From an analysis point of view, the class of P/T-nets is preferable because of the availability of efficient algorithms and powerful analysis tools.

In the process dimension, it is specified which tasks need to be executed and in what order. Modeling a workflow process definition in terms of a P/T-net is rather straightforward: *Tasks* are modeled by *transitions*, *conditions* are modeled by *places*, and *cases* are modeled by *tokens*. Consider for example Figure 2.2. The P/T-net shown specifies the processing of complaints; each case corresponds to one complaint. There are seven tasks. Each task is modeled by a transition. Place i models the condition that a new case has been created. The token in place i refers to a newly created case for which no tasks have been executed yet.

A marked, labeled P/T-net which models a workflow process definition is called a *WorkFlow net* (WF-net). A WF-net satisfies two requirements. First, a WF-net has one place i without any input transitions and one place o without output transitions. A token in i corresponds to a case which needs to be handled; a token in o corresponds to a case which has been completed. Second, in a WF-net there are no dangling tasks (transitions) and/or conditions (places). Every task and condition should contribute to the processing of cases. Therefore, every node of a WF-net should be located on a path from place i to place o. The latter requirement corresponds to strongly connectedness if place o is connected to i via an additional transition $\bar{t}$.

Definition 2.19. (WF-net) Let $N = (P, T, F, \ell)$ be an L-labeled P/T-net and $\bar{t} \in U$ a fresh identifier not in $P \cup T$. Net N is a *workflow net* (WF-net) if and only if the following conditions are satisfied:

1. *case creation*: P contains an input place $i \in U$ such that $\bullet i = \emptyset$,
2. *case completion*: P contains an output place $o \in U$ such that $o\bullet = \emptyset$, and
3. *connectedness*: $\bar{N} = (P, T \cup \{\bar{t}\}, F \cup \{(o, \bar{t}), (\bar{t}, i)\}, \ell \cup \{(\bar{t}, \tau)\})$ is strongly connected.

For any WF-net N, the extended net $\bar{N}$ used to formulate the connectedness constraint is called the *short-circuited* net. The label of the new transition in the short-circuited net is not important. For the sake of convenience, the label is set to τ.

The P/T-net shown in Figure 2.2 is a WF-net satisfying the requirements given in Definition 2.19. The reader is referred to [2] for more information on modeling workflow process definitions in terms of WF-nets.

The input place i corresponds to the initial state and the output place o corresponds to the final state, i.e., a case starts in marking $[i]$ and completes in marking $[o]$. In the previous subsection, we introduced branching bisimilarity which distinguishes successful termination and deadlock. A workflow can only terminate successfully in marking $[o]$. Therefore, for WF-nets, the termination predicate is defined as follows.

Definition 2.20. The class of marked, labeled P/T-nets $\mathcal{N}$ is equipped with the following termination predicate: $\downarrow = \{(N, [o]) \mid N \text{ is a WF-net}\}$.

Note that the fact that a WF-net contains the output place o does not necessarily mean that, in the initial marking, it has the option to terminate successfully. In the next subsection, we address among other things successful termination of workflow processes.

2.5 Soundness

As mentioned, a workflow process definition specifies the life cycle of one case in isolation. This means that we are interested in the behavior of WF-nets that have initially a single token in the special place i. The requirements given in Definition 2.19 (WF-net) only refer to the structure of the P/T-net modeling a workflow process definition. Despite these structural requirements, the behavior of a WF-net can contain problems such as deadlocks, livelocks, dangling references upon completion of a case, and tasks that can never be executed. Consider for example the WF-net shown in Figure 2.21. The WF-net describes the procedure for handling complaints. It is an extension of the WF-net shown in Figure 2.2. However, while extending this WF-net an error has been introduced. If the task *send_letter* is executed, a deadlock occurs; the system gets stuck in the marking [*pending_complaint*, *inform_man*]. The source of this problem is place *error*. This place is depicted in bold and named error to highlight the crux of the problem. Another problem occurs if *ignore_complaint* is executed; a token gets stuck in place *error* and the case completes (i.e., a token is put in place o) without removing this token. The token in *error* can be seen as a dangling reference to the already completed case. Since most workflow management systems have no garbage collection, such a completion is undesirable. Moreover, after completing a case, it should be guaranteed that no tasks are executed for this case. To avoid these, and other, problems we formulate additional requirements.

Definition 2.22. (Soundness) A WF-net N is said to be *sound* if and only if the following conditions are satisfied:

1. *safeness*: $(N, [i])$ is safe,
2. *proper completion*: for any reachable marking $s \in [N, [i]\rangle$, $o \in s$ implies $s = [o]$,
3. *absence of deadlock*: for any reachable marking $s \in [N, [i]\rangle$, $[o] \in [N, s\rangle$, and
4. *absence of dead tasks*: $(N, [i])$ contains no dead transitions.

Soundness is the minimal requirement any workflow process definition should satisfy. The first requirement in Definition 2.22 states that a sound WF-net is safe. This is a reasonable assumption since places in a WF-net correspond to conditions which are either true (marked by a token) or false (empty). The second requirement states that the moment a token is put in place $[o]$ all the other places should be empty, which corresponds to the completion of a case without dangling references. The third requirement states that starting from the initial marking $[i]$ it is always possible to reach the marking with one token in place o, which means that it is always feasible to complete a case successfully. The

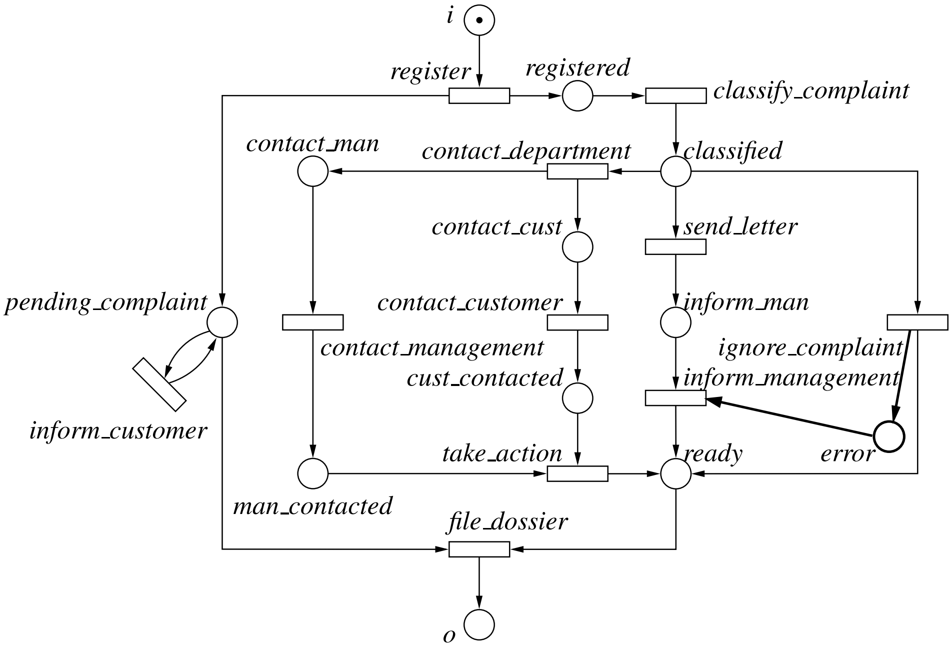

Figure 2.21: A WF-net that is not sound.

last requirement, which states that there are no dead transitions, corresponds to the requirement that for each task there is an execution of the workflow in which the task is performed.

The notion of soundness used in this paper is slightly stronger than the notion of soundness used in previous publications, i.e., the first requirement is not present in [1, 2]. The safeness requirement has been added to stress the fact that places correspond to conditions which either hold (one token) or do not hold (no tokens). In addition, the requirement allows for the simplification of the inheritance rules pivotal to this paper.

The WF-net shown in Figure 2.2 is sound. This can easily be verified by inspecting the seven reachable states. The WF-net shown in Figure 2.21 is not sound because the second requirement (proper completion: marking $[o, error]$ is reachable), the third requirement (absence of deadlock: deadlock in marking [*pending_complaint*, *inform_man*]), and the fourth requirement (absence of dead tasks: *inform_management* will never fire) of Definition 2.22 (Soundness) are not guaranteed.

The notion of soundness coincides with liveness and safeness of the short-circuited net.

Theorem 2.23. (Characterization of soundness) *A WF-net N is sound if and only if $(\bar{N}, [i])$ is live and safe.*

Proof. The proof is similar to the proof of Theorem 11 in [1]. The only difference is that in this paper a stronger notion of soundness is used, which implies safeness rather than boundedness of the short-circuited net. □

This theorem shows that standard Petri-net-based analysis techniques can be used to verify soundness. Consider for example the WF-net shown in Figure 2.2. We can use one of the many standard Petri-net-based analysis tools (cf. [21]) to verify that the short-circuited net is live and safe. The exact complexity of deciding whether a WF-net is sound is not known though it is very likely – and in the worst case – PSPACE-complete (see [29]). A very straightforward approach to deciding soundness is the construction of a coverability graph (see, for example, [53]) of the short-circuited net in its initial marking. This approach requires, in the worst case, non-primitive recursive space. However, most workflow management systems use a modeling language which corresponds to free-choice P/T-nets ([2]). For free-choice WF-nets, soundness can be decided in polynomial time ([1]). Practical experience with workflow management systems that allow for the design of non-free-choice WF-nets (e.g., COSA) shows that even the more complex workflows have less than 100.000 states and can be checked using tools that are based on the coverability-graph algorithm such as Woflan [59]. Woflan is briefly described in Section 7; for more information, the interested reader is referred to [58, 59].

Definition 2.24. (Workflow process definition) A *workflow process definition* is a sound WF-net. The set of all workflow process definitions is denoted $\mathcal{W}$.

Class $\mathcal{W}$ is the class of labeled P/T-nets that is interesting in the context of workflow management. Members of this class are called *workflow process definitions* and are guaranteed to be correct with respect to the criteria mentioned in Definitions 2.19 (WF-net) and 2.22 (Soundness). Note that, formally, a workflow process definition is defined as a P/T net *without* an initial marking. However, in the remainder, we typically consider markings reachable from the initial marking $[i]$. Therefore, if no initial marking is given explicitly, the initial marking of a workflow process definition is assumed to be $[i]$.

In this paper, branching bisimilarity is used as a behavioral equivalence relation. Therefore, in the remainder, we assume that branching bisimilarity is the standard equivalence relation for comparing workflow process definitions.

Definition 2.25. (Behavioral equivalence of workflow process definitions) For any two workflow process definitions N_0 and N_1 in $\mathcal{W}$, $N_0 \cong N_1$ if and only if $(N_0, [i]) \sim_b (N_1, [i])$.

3 Inheritance

Inheritance is one of the cornerstones of object-oriented programming and object-oriented design. The basic idea of inheritance is to provide mechanisms which allow for constructing *subclasses* that inherit certain properties of a given *superclass*. In our case, a *class* corresponds to a *workflow process definition* (i.e., a sound WF-net; see Definition 2.24) and *objects* (i.e., instances of the class) correspond to *cases*. In most object-oriented design methods, a class is characterized by a set of *attributes* and a set of *methods*. Attributes are used to describe properties of an object. Methods specify operations on objects (e.g., create, destroy, and change attribute). Note that attributes and methods only describe the static aspects of an object. The dynamic behavior of an object is either hidden inside the methods or modeled explicitly. (In UML [19], the behavior of an object is modeled in terms of a state machine.) Although the dynamic behavior of objects is an intrinsic part of the class description (either explicit or implicit), inheritance of dynamic behavior is not well-understood. (See [15] for an elaborate discussion on this topic and pointers to related work.) Since every object-oriented programming language supports inheritance with respect to the static structure of a class (i.e., the interface consisting of attributes and methods), this is remarkable. Since workflow management aims at supporting business processes, results on inheritance of static aspects are not very useful in this context. However, we can use the work presented in [15, 14, 4, 16] where inheritance of dynamic behavior is dealt with in a comprehensive manner. Other approaches either focus on very specific inheritance relations or abstract from the causal relations between tasks/methods. Consider for example the work by Malone et al. [45] where inheritance is defined for tasks and processes. They also provide tool support for navigating through a space of processes using specialization and generalization links. Unfortunately, the control or routing structure is not taken into account, i.e., causal relations between tasks are not considered. Some of the workflow management systems available claim to be object-oriented and thus provide some support for inheritance. For example, the workflow management system InConcert [39] allows for building workflow class hierarchies. Unfortunately, inheritance is restricted to attributes and the structure of a process is not taken into account. Many workflow management systems have been implemented using object-oriented programming languages. However, these systems do not offer object-oriented mechanisms such as inheritance to the workflow designer or the designer has to program code to benefit from the object-oriented features provided by the host language. Nevertheless, we think that inheritance is a very useful concept for workflow management. Therefore, we advocate the use of the inheritance notions presented in [15, 14, 4, 16] and illustrate the usefulness by tackling the problems related to change.

3.1 Inheritance relations

In this subsection, we define four inheritance relations for workflow processes. Consider two workflow process definitions x and y in $\mathcal{W}$. When is x a subclass of y? Process definition x is a subclass of superclass y if x inherits certain features of y. Intuitively, one could say that x is a subclass of y if and only if x can do what y can do. Clearly, all tasks present in y should also be present in x. Moreover, x will typically add new tasks. Therefore, it is reasonable to demand that x can do what y can do with respect to the tasks present in y. With respect to new tasks (i.e., tasks present in x but not in y), there are basically two mechanisms which can be used. The first mechanism simply disallows the execution of any new tasks and then compares the resulting behavior of x with the behavior of y. This mechanism leads to the following notion of inheritance.

> *If it is not possible to distinguish the behaviors of x and y when only tasks of x that are also present in y are executed, then x is a subclass of y.*

Intuitively, this definition conforms to *blocking* tasks new in x. The resulting inheritance concept is called *protocol inheritance*; x inherits the protocol of y.

Another mechanism would be to allow for the execution of new tasks but to consider only the effects of old ones.

> *If it is not possible to distinguish the behaviors of x and y when arbitrary tasks of x are executed, but when only the effects of tasks that are also present in y are considered, then x is a subclass of y.*

This inheritance notion is called *projection inheritance*; x inherits the projection of workflow process definition y onto the old tasks. Projection inheritance conforms to *hiding* or *abstracting from* tasks new in x.

Recall from Section 2.3 that branching bisimilarity is the equivalence used to compare the behaviors of marked P/T-nets and, thus, the behaviors of workflow process definitions. Also recall that the action label τ is used to denote internal or unobservable actions. As a consequence, hiding tasks in a workflow process definition can be achieved by renaming these tasks to τ. In the remainder of this paper, we assume that the set of action or task labels L used in WF-nets (see Definition 2.19) is equal to the set O of observable tasks extended with τ, i.e., $L = \{\tau\} \cup O$.

Although the distinction between the two inheritance mechanisms presented above may seem subtle, the corresponding inheritance notions are quite different. To illustrate this difference, we use the

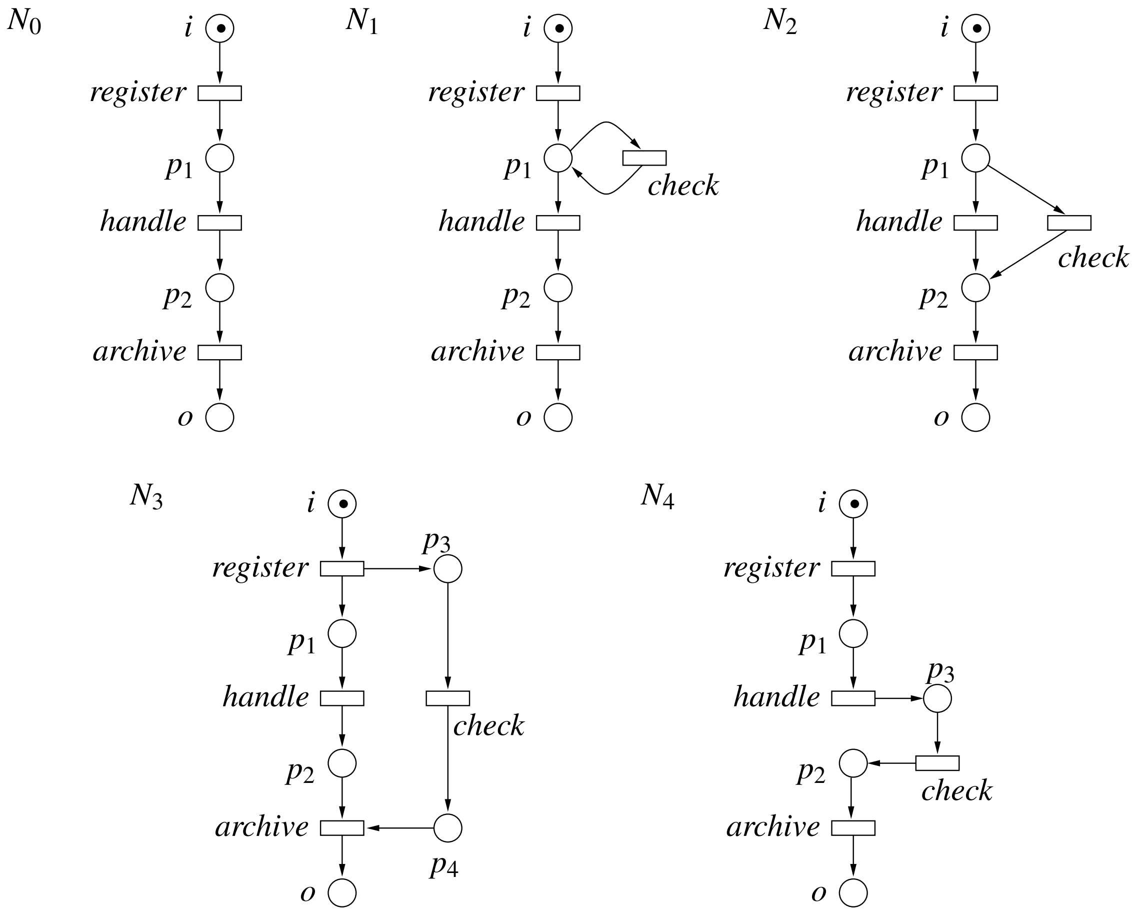

Figure 3.1: Five workflow process definitions.

five workflow process definitions shown in Figure 3.1. Process definition N_0 corresponds to a sequential workflow process which consists of three tasks: *register*, *handle*, and *archive*. Each of the other workflow process definitions (i.e., N_1, N_2, N_3, and N_4) extends N_0 with an additional task *check*. In process definition N_1, task *check* can be executed arbitrarily many times between *register* and *handle*. Process N_1 is a subclass of N_0 with respect to protocol inheritance; if *check* is blocked, then N_1 is identical to N_0. Process N_1 is also a subclass of N_0 with respect to projection inheritance; if every execution of *check* is hidden, then N_1 is equivalent (as defined in Definition 2.25) to N_0. In N_2, task *check* can be executed instead of task *handle*. Process N_2 is a subclass of N_0 with respect to protocol inheritance; if *check* is blocked, then N_2 is equivalent to N_0. Process definition N_2 is not a subclass of N_0 with respect to projection inheritance, because it is possible to skip task *handle* by executing the (hidden) task *check*. In process definition N_3, task *check* is executed in parallel with task *handle*. Process N_3 is not a subclass of N_0 with respect to protocol inheritance; if *check* is blocked, then task *archive* cannot be executed. However, N_3 is a subclass of N_0 with respect to projection inheritance. If one abstracts from the newly added parallel task *check*, one cannot distinguish N_3 and N_0. Task *check* is inserted between *handle* and *archive* in the remaining workflow process definition N_4 shown in Figure 3.1. Process N_4 is not a subclass of N_0 with respect to protocol inheritance; if *check* is blocked, then the process deadlocks after executing task *handle*. However, N_4 is a subclass of N_0 with respect to projection inheritance. If one abstracts from *check*, one cannot observe any differences between the behaviors of N_4 and N_0.

The two mechanisms (i.e., blocking and hiding) result in two orthogonal inheritance notions. Therefore, we also consider combinations of the two mechanisms. A workflow process definition is a subclass of another workflow process definition under *protocol/projection inheritance* if and only if both by hiding the new methods and by blocking the new methods one cannot detect any differences, i.e., it is a subclass under both protocol and projection inheritance. In Figure 3.1, N_1 is a subclass of N_0 with respect to protocol/projection inheritance. The two mechanisms can also be used to obtain a weaker form of inheritance. A workflow process definition is a subclass of another workflow process definition under *life-cycle inheritance* if and only if by blocking some newly added tasks and by hiding some others one cannot distinguish between them. Life-cycle inheritance is more general than the other three inheritance relations. All workflow process definitions shown in Figure 3.1 are subclasses of N_0 with respect to life-cycle inheritance. A detailed study of the four inheritance relations can be found in [15, 14]. For the purpose of this paper, it suffices to formalize the relations. We do not go into much detail about the properties of the inheritance relations.

To formalize the four forms of inheritance, we introduce two operators on P/T-nets, namely encapsulation and abstraction. Encapsulation is used to block tasks; abstraction is used to hide tasks. The two operators are inspired by the encapsulation and abstraction operators known in process algebra [12]. The operators can be defined on labeled P/T-nets as follows.

Definition 3.2. (Encapsulation) Let $N = (P, T_0, F_0, \ell_0)$ be an L-labeled P/T-net. For any $H \subseteq O$, the encapsulation operator ∂_H is a function that removes from a given P/T-net all transitions with a label in H. Formally, $\partial_H(N) = (P, T_1, F_1, \ell_1)$ such that $T_1 = \{t \in T_0 \mid \ell_0(t) \notin H\}$, $F_1 = F_0 \cap ((P \times T_1) \cup (T_1 \times P))$, and $\ell_1 = \ell_0 \cap (T_1 \times L)$.

Note that removing transitions from a WF-net as defined in Definition 2.19 might yield a result that is no longer a WF-net.

Definition 3.3. (Abstraction) Let $N = (P, T, F, \ell_0)$ be an L-labeled P/T-net. For any $I \subseteq O$, the abstraction operator τ_I is a function that renames all transition labels in I to the silent action τ. Formally, $\tau_I(N) = (P, T, F, \ell_1)$ such that, for any $t \in T$, $\ell_0(t) \in I$ implies $\ell_1(t) = \tau$ and $\ell_0(t) \notin I$ implies $\ell_1(t) = \ell_0(t)$.

Given these two operators, the four notions of inheritance can be defined as follows:

Definition 3.4. (Inheritance relations)

1. *Protocol inheritance*:
   For any workflow process definitions N_0 and N_1 in $\mathcal{W}$, workflow process definition N_1 is a subclass of N_0 under protocol inheritance, denoted $N_1 \leq_{pt} N_0$, if and only if there is an $H \subseteq O$ such that $(\partial_H(N_1), [i]) \sim_b (N_0, [i])$.

2. *Projection inheritance*:
   For any workflow process definitions N_0 and N_1 in $\mathcal{W}$, workflow process definition N_1 is a subclass of N_0 under projection inheritance, denoted $N_1 \leq_{pj} N_0$, if and only if there is an $I \subseteq O$ such that $(\tau_I(N_1), [i]) \sim_b (N_0, [i])$.

3. *Protocol/projection inheritance*:
   For any workflow process definitions N_0 and N_1 in $\mathcal{W}$, workflow process definition N_1 is a subclass of N_0 under protocol/projection inheritance, denoted $N_1 \leq_{pp} N_0$, if and only if there is an $H \subseteq O$ such that $(\partial_H(N_1), [i]) \sim_b (N_0, [i])$ *and* an $I \subseteq O$ such that $(\tau_I(N_1), [i]) \sim_b (N_0, [i])$.

4. *Life-cycle inheritance*:
   For any workflow process definitions N_0 and N_1 in $\mathcal{W}$, workflow process definition N_1 is a subclass of N_0 under life-cycle inheritance, denoted $N_1 \leq_{lc} N_0$, if and only if there are an $I \subseteq O$ and an $H \subseteq O$ such that $I \cap H = \emptyset$ and $(\tau_I \circ \partial_H(N_1), [i]) \sim_b (N_0, [i])$.

$$\begin{tikzcd}
 & \leq_{pp} \arrow[dl] \arrow[dr] & \\
\leq_{pt} \arrow[dr] & & \leq_{pj} \arrow[dl] \\
 & \leq_{lc} &
\end{tikzcd}$$

Figure 3.5: An overview of the four inheritance relations for behavior.

The four inheritance relations are based on the equivalence notion (branching bisimilarity) introduced in Definition 2.16. Note that for life-cycle inheritance the new tasks are partitioned into two sets H and I: Tasks that are blocked by means of the operator ∂_H and tasks that are hidden by means of τ_I. Figure 3.5 gives an overview of the four inheritance relations. The arrows depict strict inclusion relations. It is easy to see that protocol/projection inheritance implies both protocol and projection inheritance. Moreover, protocol inheritance implies life-cycle inheritance and also projection inheritance implies life-cycle inheritance. However, life-cycle inheritance does not imply protocol or projection inheritance. Consider for example the workflow process definition shown in Figure 3.6. This workflow process definition extends the process definition shown in Figure 2.2 with four new tasks: *inform_customer*, *contact_management*, *inform_management*, and *ignore_complaint*. It corresponds to the WF-net of Figure 2.21 without the place *error*. (Note that in contrast to the WF-net of Figure 2.21 the soundness property is satisfied.) The question is whether the extended workflow process definition shown in Figure 3.6 is a subclass of the workflow process definition shown in Figure 2.2. It is not a subclass under protocol inheritance; blocking *contact_management* results in a potential deadlock. It is also not a subclass under projection inheritance; by executing *ignore_complaint*, the original task *send_letter* is skipped. Since protocol/projection inheritance requires both protocol inheritance and projection inheritance, the extended workflow process definition is clearly not a subclass under protocol/projection inheritance. However, the extended workflow process definition shown in Figure 3.6 is a subclass of the workflow process definition of Figure 2.2 under life-cycle inheritance; by hiding *contact_management* and *inform_management*, blocking *ignore_complaint*, and hiding or blocking *inform_customer*, one obtains a workflow process definition that is branching bisimilar to the original one.

The four inheritance relations introduced in this subsection have a number of desirable properties. For example, the relations are preorders (i.e., they are reflexive and transitive; see Property 6.21 in [15]). Furthermore, if one workflow process definition is a subclass of another workflow process definition under any of the four inheritance relations and vice versa, then the two workflow process definitions are equivalent as defined in Definition 2.25 (i.e., the two workflow process definitions are branching bisimilar; see Property 6.23 in [15]). In other words, the four inheritance relations are anti-symmetric. A relation that is reflexive, anti-symmetric, and transitive is a partial order. Thus, the following property is given without further proof.

Property 3.7. *Assuming $\cong$, as defined in Definition 2.25, as the equivalence on workflow process definitions, $\leq_{lc}$, $\leq_{pt}$, $\leq_{pj}$, and $\leq_{pp}$ are partial orders.*

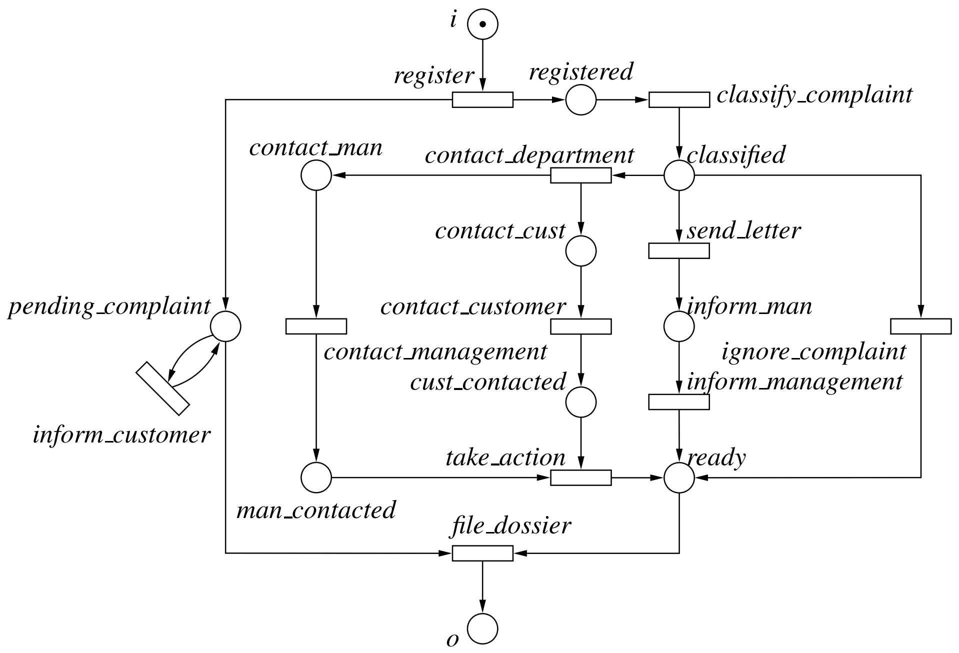

Figure 3.6: An extended workflow process definition.

Another observation is that the definition of life-cycle inheritance does not allow that some executions of a task are blocked while other executions of *the same task* are hidden or left untouched in determining a subclass relationship between two workflow process definitions. To illustrate this restriction, consider a workflow process definition N_5. Process definition N_5 is an extension of workflow process definition N_0 of Figure 3.1 that combines the two extensions of process definitions N_2 and N_3 in the same figure. Process definition N_5 is not a subclass under life-cycle inheritance of N_1, whereas the workflow process definitions N_2 and N_3 are. The reason is that life-cycle inheritance does not allow the encapsulation of task *check* when it is executed as an alternative to task *handle* and the abstraction of task *check* when it is executed in parallel to *handle*. However, it is not difficult to generalize the definition of life-cycle inheritance, or any of the other three inheritance relations for that matter, in such a way that it is allowed to treat different executions of the same task in a different way. It simply requires the use of temporary task names to distinguish the different executions of a single task. For example, such a variant of life-cycle inheritance could be defined as follows. If a workflow process definition is a subclass under the current definition of life-cycle inheritance of another workflow process definition, then any renaming of the tasks new in the subclass yields a subclass under the variant of life-cycle inheritance. Consider, for example, the variant N_6 of the workflow process definition N_5 introduced above in which the two checks have names $check_2$ and $check_3$, respectively. It is not difficult to see that blocking one of these tasks and hiding the other one proves that N_6 is a subclass of N_0 under the current definition of life-cycle inheritance. Renaming the two tasks $check_2$ and $check_3$ to *check* proves that N_5 is a subclass of N_0 under the proposed variant of life-cycle inheritance. However, in this paper, we do not formalize the generalizations of the four inheritance relations along the lines discussed in this paragraph. The goal is to focus on the important concepts that play a role when applying inheritance notions in the context of workflow management. Although the above generalizations might be useful in some occasions, they distract from the essential concepts.

Finally, the question remains which inheritance relation is appropriate. The answer to this question depends on the context. For some applications, a very liberal notion of inheritance is suitable (i.e., life-cycle inheritance). For other applications, a more restrictive notion is desirable. In Section 4, we discuss the usefulness of the inheritance relations in different application areas in the context of workflow management.

3.2 Inheritance-preserving transformation rules

The inheritance relations of the previous subsection by themselves are not always immediately useful. The workflow designer can only benefit from the inheritance relations if there is a method or a tool to support workflow changes which preserve inheritance. For this purpose, we present four inheritance-preserving transformation rules. Each of these transformation rules can be used to construct a subclass of a given workflow process definition by extending it. The rules are local and relatively easy to check (from a computational point of view). Furthermore, they correspond to typical design constructs used by a workflow designer to extend or change a workflow.

The rules presented in this paper are slightly different versions of the rules presented in [15, 14, 4, 16]. The main distinction is the requirement that workflow process definitions (called life cycles in [15, 14, 4, 16]) have to be safe. Therefore, the rules are named *PPS*, *PTS*, *PJS*, and *PJ3S* rather than *PP*, *PT*, *PJ*, and *PJ3*. The safeness requirement simplifies the formulation of the rules and allows for generalizations with respect to the free-choice requirements stated in [15].

Two auxiliary definitions are needed for the definition of the transformation rules.

Definition 3.8. (Alphabet) The alphabet operator is a function $\alpha : \mathcal{N} \to \mathcal{P}(O)$. Let (N, s) be a marked, L-labeled P/T-net in $\mathcal{N}$, with $N = (P, T, F, \ell)$. The alphabet of (N, s) is defined as the set of visible labels of all transitions of the net that are not dead: $\alpha(N, s) = \{\ell(t) \mid t \in T \wedge \ell(t) \neq \tau \wedge t$ is not dead in $(N, s)\}$.

Since workflow process definitions do not contain dead transitions, the alphabet of a workflow process definition equals the set of its observable transition labels.

Property 3.9. (Alphabet of a workflow process definition) *For any $N = (P, T, F, \ell) \in \mathcal{W}$, the alphabet $\alpha(N, [i])$ equals $\{\ell(t) \mid t \in T \wedge \ell(t) \neq \tau\}$.*

Proof. It follows immediately from Definitions 2.24 (Workflow process definition), 2.22 (Soundness), and 3.8 (Alphabet). □

For the sake of simplicity, the alphabet of a workflow process definition $N \in \mathcal{W}$ is denoted $\alpha(N)$.

Definition 3.10. (Union of labeled P/T-nets) Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$ be two L-labeled P/T-nets such that $(P_0 \cup P_1) \cap (T_0 \cup T_1) = \emptyset$ and such that, for all $t \in T_0 \cap T_1$, $\ell_0(t) = \ell_1(t)$. The union $N_0 \cup N_1$ of N_0 and N_1 is the labeled P/T-net $(P_0 \cup P_1, T_0 \cup T_1, F_0 \cup F_1, \ell_0 \cup \ell_1)$. If two P/T-nets satisfy the abovementioned two conditions, their union is said to be *well defined*.

The rule that is the easiest one to understand is presented first. It is named *PPS* and preserves both protocol and projection inheritance. Transformation rule *PPS* is illustrated in Figure 3.11. Let N_0 be a workflow process definition. Let N be a (connected) P/T-net such that the union $N_1 = N_0 \cup N$ is well defined. The workflow process definition N_1 is a subclass of process definition N_0 under protocol/projection inheritance if the following four conditions are satisfied: (1) N_0 and N only share a single place p, (2) all transitions of N have a label which does not appear in the alphabet of N_0, (3) each transition of N with p as one of its input places has a *visible* label, and (4) $(N, [p])$ is live and safe. Transformation rule *PPS* shows that under protocol/projection inheritance, it is allowed to *postpone* behavior. When $(N_1, [i])$ reaches a state in which place p is marked, it is possible to iterate the behavior defined by N an arbitrary number of times before continuing with the original behavior. The requirement that $(N, [p])$ is live and safe guarantees that every token consumed from place p by a transition of N can always be returned to p. This property of N is crucial for the correctness of rule *PPS*.

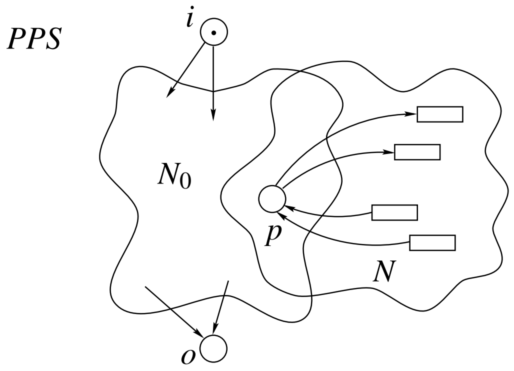

Figure 3.11: A protocol/projection-inheritance-preserving transformation rule.

Theorem 3.12. (Protocol/projection-inheritance-preserving transformation rule *PPS***)** *Let* $N_0 = (P_0, T_0, F_0, \ell_0)$ *be a workflow process definition in* $\mathcal{W}$. *If* $N = (P, T, F, \ell)$ *is a labeled P/T-net with place* $p \in P$ *such that*

1. $p \notin \{i, o\}$, $P_0 \cap P = \{p\}$, $T_0 \cap T = \emptyset$,
2. $(\forall t : t \in T : \ell(t) \notin \alpha(N_0))$,
3. $(\forall t : t \in T \wedge p \in \bullet t : \ell(t) \neq \tau)$,
4. $(N, [p])$ *is live and safe, and*
5. $N_1 = N_0 \cup N$ *is well defined,*

then N_1 *is a workflow process definition in* $\mathcal{W}$ *such that* $N_1 \leq_{pp} N_0$.

Proof. Transformation rule *PPS* is a special case of two rules that are presented in the remainder, namely the rules *PTS* of Theorem 3.14 and *PJS* of Theorem 3.16. Since *PTS* preserves protocol inheritance and *PJS* preserves projection inheritance, it is shown that rule *PPS* preserves both protocol and projection inheritance and, thus, protocol/projection inheritance. The proof is a simplification of the proof of Theorem 7.3 in [15]. (The free-choice requirement in [15] is replaced by the condition that both N_0 and N are safe.) □

In Figure 3.1, N_1 can be constructed from N_0 using transformation rule *PPS*; place p_1 is the place shared by N_0 and the extension containing transition *check*.

The remaining three transformation rules of this subsection are all based on the same principles as rule *PPS*. The second transformation rule of this subsection, named *PTS*, preserves protocol inheritance. It is illustrated in Figure 3.13. Transformation rule *PTS* can be used to extend a given workflow process definition with alternative branches of behavior. Let N_0 be a workflow process definition. The extension of N_0 is based on a P/T net N with a place p_i such that $(N, [p_i])$ is live and safe. Nets N_0 and N share two places p_i and p_o and no other nodes. Furthermore, N contains a transition y with p_o as its only input place and p_i as its only output place. The P/T net N_1 resulting from transformation rule *PTS* is defined as the union of N_0 and N after the removal of transition y. Place p_i functions as the entry point of the alternative branches of behavior added to N_0, whereas p_o functions as the exit point. The requirement that $(N, [p_i])$ is live and safe ensures that any token that transitions in N consume from place p_i is eventually returned to p_o. Two additional requirements guarantee that N_1 is a workflow process definition. First, it is required that N_0 extended with a fresh transition x with input place p_i and output place p_o is a workflow process definition. Transition x emulates the behavior of N in N_1. Note that x is only introduced to formulate the requirements of rule *PTS*; it is not present in the original process definition N_0, the extension N, or the subclass N_1. Second, transformation rule *PPS* is a special case of transformation rule *PTS*. Rule *PTS* reduces to rule *PPS* when places p_i and p_o coincide. If places p_i and p_o are different, then it is assumed that the only input transition of place p_i in N is transition y and that the only output transition of p_o in N is y. This assumption excludes the possibility of iterations beginning and ending in place p_i or place p_o, thus guaranteeing that the modification of the original process definition truly has place p_i as its entry point and place p_o as its exit point. A final requirement guarantees that N_1 is a subclass of N_0 under protocol inheritance: All transitions of N with input place p_i must have a visible label not appearing in the alphabet of N_0. This requirement means that transitions of N with input places in N_0 act as so-called *guards*. Encapsulating the guards leads to a net whose behavior is identical to the behavior of the original process definition, thus guaranteeing that N_1 is a subclass of N_0 under protocol inheritance.

Recall that we use $\overset{N}{\bullet}\_$ and $\_\overset{N}{\bullet}$ to denote the preset and postset functions of N. Without this notation it is not possible to distinguish the preset and postset functions of the extension N from those of the original workflow process definition N_0 and the resulting subclass N_1.

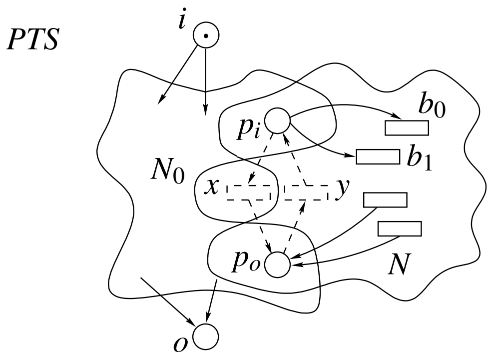

Figure 3.13: A protocol-inheritance-preserving transformation rule.

Theorem 3.14. (Protocol-inheritance-preserving transformation rule *PTS***)** *Let* $N_0 = (P_0, T_0, F_0, \ell_0)$ *be a workflow process definition in* $\mathcal{W}$*. Let* $N = (P, T, F, \ell)$ *be a labeled P/T net. Assume that* $x \in U$ *is a fresh identifier not appearing in* $P_0 \cup T_0 \cup P \cup T$*. If* N *contains places* $p_i, p_o \in P$ *and a transition* $y \in T$ *such that*

1. $P_0 \cap P = \{p_i, p_o\}$, $T_0 \cap T = \emptyset$,

2. $\overset{N}{\bullet} y = \{p_o\}$, $y\overset{N}{\bullet} = \{p_i\}$, $p_i \neq p_o \Rightarrow \overset{N}{\bullet} p_i = p_o\overset{N}{\bullet} = \{y\}$,

3. $(\forall t : t \in p_i \overset{N}{\bullet} : \ell(t) \in O \backslash \alpha(N_0))$,

4. $(N, [p_i])$ *is live and safe,*

5. $N_1 = N_0 \cup (P, T\backslash\{y\}, F\backslash\{(y, p_i), (p_o, y)\}, \ell\backslash\{(y, \ell(y))\})$ *is well defined, and*

6. $N_0^x = (P_0, T_0 \cup \{x\}, F_0 \cup \{(p_i, x), (x, p_o)\}, \ell_0 \cup \{(x, \tau)\})$ *is a workflow process definition,*

then N_1 *is a workflow process definition in* $\mathcal{W}$ *such that* $N_1 \leq_{pt} N_0$.

Proof. The proof is similar to the proof of Theorem 7.17. in [15]. (The free-choice requirement in [15] is replaced by the condition that both N_0 and N are safe.) □

To illustrate transformation rule *PTS*, we use the workflow process definitions shown in Figure 3.1. Process definition N_1 can be constructed from N_0 using transformation rule *PTS*; net N is the net containing place p_1 and transition *check*. Note that, in this particular case, p_i and p_o coincide. Net N_2 can also be constructed from N_0 using transformation rule *PTS*. Since the remaining workflow process definitions (i.e., N_3 and N_4) are no subclasses of N_0 with respect to protocol inheritance, it makes no sense to try and apply *PTS* to obtain either of these workflow process definitions.

The next transformation rule of this subsection, *PJS*, preserves projection inheritance. Theorem 3.16 given below formalizes transformation rule *PJS*. Figure 3.15 illustrates the rule. It shows that rule *PJS* corresponds to a sequential composition. New behavior may be inserted between sequential parts of a workflow process definition, yielding a subclass under projection inheritance. In contrast to the previous two transformation rules, the original workflow process definition is modified. Basically, transformation rule *PJS* says that it is allowed to replace an arc in the original workflow process definition by an entire P/T-net. The original workflow process definition N_0 contains a place p which has a transition t_p as one of its input transitions. The modification of N_0 is based upon a P/T-net N sharing place p and transition t_p with N_0. Place p is the only input place of t_p in N. The result of the transformation rule is the P/T-net N_1 obtained by taking the union of N_0 and N after removing both the arc between t_p and p from N_0 and the arc between p and t_p from N. The requirement that $(N, [p])$ is live and safe guarantees that N_1 always has the option to move every token that transition t_p would normally have put into place p to place p by only firing transitions of N. The requirement that all transitions of N other than t_p are labeled with task identifiers not appearing in the alphabet of N_0 guarantees that hiding these tasks does not influence the behavior of the original workflow process definition.

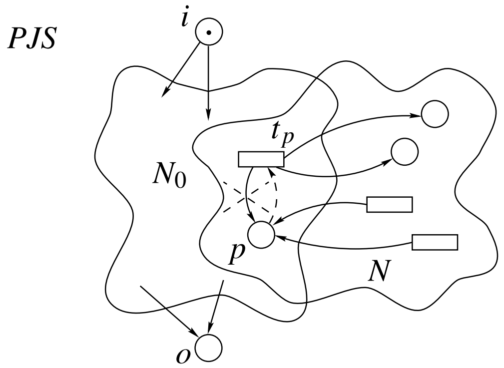

Figure 3.15: A projection-inheritance-preserving transformation rule.

Theorem 3.16. (Projection-inheritance-preserving transformation rule *PJS*) *Let* $N_0 = (P_0, T_0, F_0, \ell_0)$ *be a workflow process definition in* $\mathcal{W}$*. If* $N = (P, T, F, \ell)$ *is a labeled P/T-net with place* $p \in P$ *and transition* $t_p \in T$ *such that*

1. $P_0 \cap P = \{p\}$, $T_0 \cap T = \{t_p\}$, $(t_p, p) \in F_0$, $\overset{N}{\bullet} t_p = \{p\}$, *and* $p = o \Rightarrow p \overset{N}{\bullet} = \{t_p\}$,
2. $(\forall t : t \in T \backslash T_0 : \ell(t) \notin \alpha(N_0))$,
3. $(N, [p])$ *is live and safe, and*
4. $N_1 = (P_0, T_0, F_0 \backslash \{(t_p, p)\}, \ell_0) \cup (P, T, F \backslash \{(p, t_p)\}, \ell)$ *is well defined,*

then N_1 *is a workflow process definition in* $\mathcal{W}$ *such that* $N_1 \leq_{pj} N_0$.

Proof. The proof is similar to the proof of Theorem 7.13 in [15]. (The free-choice requirement in [15] is replaced by the condition that both N_0 and N are safe.) $\square$

In Figure 3.1, N_4 can be constructed from N_0 using transformation rule *PJS*; the arc between transition *handle* and place p_2 is replaced by the net containing place p_3 and transition *check*.

As mentioned in the proof of Theorem 3.12, transformation rule *PPS* of Theorem 3.12 is a special case of transformation rule *PJS* of Theorem 3.16. For more details, the reader is referred to [15]. (It is an interesting exercise to prove this claim.)

To formulate the last transformation rule of this subsection, the following auxiliary definition is needed. A place of a marked P/T-net is said to be *redundant* or *implicit* if and only if it does not depend on the number of tokens in the place whether any of its output transitions is enabled by some reachable marking.

Definition 3.17. (Implicit place) Let (N, s) with $N = (P, T, F, \ell)$ be a marked, labeled P/T-net. A place $p \in P$ is called *implicit* in (N, s) if and only if, for any reachable markings $s' \in [N, s\rangle$ and any transition $t \in p\bullet$, $s' \geq \bullet t \backslash \{p\} \Rightarrow s' \geq \bullet t$.

Implicit places and their properties have been studied in [17, 22].

Transformation rule *PJ3S* is formalized in Theorem 3.19 given below. It shows under what restrictions it is allowed to extend a workflow process definition with a parallel branch of behavior. The result of rule *PJ3S* is a subclass of the original workflow process definition under projection inheritance. It is illustrated in Figure 3.18. As before, N_0 is the original workflow process definition. Again, the modification of N_0 is based on a P/T-net N containing a place p such that $(N, [p])$ is live and safe. The two net structures N_0 and N share two transitions t_i and t_o. In N, place p is the only input place of t_i and the only output place of t_o. Furthermore, p has no other input or output transitions. The net structure N_1 resulting from transformation rule *PJ3S* is defined as the union of N_0 and N after the removal of place p. These assumptions mean that transitions t_i and t_o function as the input and output transition of the extra parallel branch modeled by N. The basic idea is that the P/T-net $(N_1, [i])$ satisfies the property that every firing of transition t_i is eventually followed by a firing of transition t_o. The requirement that $(N, [p])$ is live and safe guarantees that each time transition t_i fires the resulting tokens in places of N can be moved to the input places of transition t_o in N by only firing transitions of N other than t_i and t_o. In addition, to guarantee that $(N_1, [i])$ satisfies the desired property, also $(N_0, [i])$ must be such that every firing of t_i is followed by exactly one firing of t_o. To achieve this goal, assume that N_0 is extended with a place q with t_i as its only input transition and t_o as its only output transition. Requiring that place q is implicit in this extension guarantees that a firing of transition t_o is always preceded by a firing of t_i. It is not difficult to see that the number of tokens in q (zero or one) corresponds to the number of firings of transition t_i which have not yet been followed by a firing of t_o. To guarantee that $(N_0, [i])$ cannot terminate without firing t_o as many times as t_i, the extension of N_0 with place q must be such that it cannot put a token in place o while leaving tokens in q. Clearly, this is achieved when the extension of N_0 with q yields another workflow process definition. The combination of the requirements on N_0 and N implies that N_1 is a workflow process definition satisfying the property that every firing of t_i is eventually followed by a firing of t_o. The attentive reader might notice the duality between rules *PTS* and *PJ3S*.

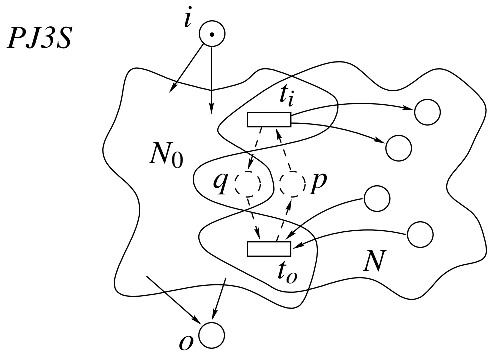

Figure 3.18: A projection-inheritance-preserving transformation rule.

Theorem 3.19. (Projection-inheritance-preserving transformation rule *PJ3S***)** *Let* $N_0 = (P_0, T_0, F_0, \ell_0)$ *be a workflow process definition in* $\mathcal{W}$. *Let* $N = (P, T, F, \ell)$ *be a labeled P/T-net. Assume that* $q \in U$ *is a fresh identifier not appearing in* $P_0 \cup T_0 \cup P \cup T$. *If* N *contains a place* $p \in P$ *and transitions* $t_i, t_o \in T$ *such that*

1. $P_0 \cap P = \emptyset$, $T_0 \cap T = \{t_i, t_o\}$,
2. $\overset{N}{\bullet} p = \{t_o\}$, $p \overset{N}{\bullet} = \{t_i\}$, $\overset{N}{\bullet} t_i = \{p\}$, $t_o \overset{N}{\bullet} = \{p\}$,
3. $(\forall t : t \in T \backslash T_0 : \ell(t) \notin \alpha(N_0))$,
4. $(N, [p])$ *is live and safe,*
5. $N_1 = N_0 \cup (P \backslash \{p\}, T, F \backslash \{(p, t_i), (t_o, p)\}, \ell)$ *is well defined,*
6. q *is implicit in* $(N_0^q, [i])$ *with* $N_0^q = (P_0 \cup \{q\}, T_0, F_0 \cup \{(t_i, q), (q, t_o)\}, \ell_0)$, *and*
7. N_0^q *is a workflow process definition,*

then N_1 *is a workflow process definition in* $\mathcal{W}$ *such that* $N_1 \leq_{pj} N_0$.

Proof. The proof is similar to the proof of Theorem 7.23 in [15]. (The free-choice requirement in [15] is replaced by the condition that both N_0 and N are safe.) □

The transformation rule defined in Theorem 3.19 is named *PJ3S* for historical reasons (see [15]). It is easy to see that workflow net N_3 in Figure 3.1 can be constructed from N_0 using transformation rule *PJ3S*.

In Section 3.1, we concluded that the workflow process definition shown in Figure 3.6 is a subclass of the workflow process definition shown in Figure 2.2 with respect to life-cycle inheritance. It is not difficult to see that the transformation rules presented in this section can be used to construct the subclass of Figure 3.6 from the superclass of Figure 2.2. Transformation rule *PTS* can be used to add the alternative task *ignore_complaint*. Rule *PJS* can be used to add task *inform_management* in-between *send_letter* and *ready*. The parallel task *contact_management* can be added using transformation rule *PJ3S*. Task *inform_customer* can be added using either *PPS*, *PTS*, or *PJS*.

In this subsection, four transformation rules have been presented to construct subclasses of workflow process definitions under different forms of inheritance. The rules correspond to design constructs that are often used in practice, namely choice, iteration, sequential composition, and parallel composition. If a designer sticks to these rules, inheritance is guaranteed. In the remainder, we show that the transformation rules can be used to avoid the problems discussed in the introduction. The rules are also interesting from a computational point of view. By using the inheritance-preserving transformation rules rather than making arbitrary changes, the complexity of checking whether the extended workflow process definition is a subclass of the original process definition is reduced considerably. Note that all requirements specified for the transformation rules can be verified locally, i.e., each requirement which involves the evaluation of dynamic behavior is a constraint on either the original workflow process definition N_0 or the extension N. For none of the rules, it is required to verify the dynamic behavior (e.g., liveness, safeness, and reachability) of the combined net N_1. Soundness of the subclass N_1 follows from local requirements. Nevertheless, the complexity of many of the requirements in the transformation rules appears to be PSPACE-complete (see [29]). From a practical point of view, this is not an unconquerable problem. The requirements are of the same complexity as checking soundness. As argued in Section 2.5, existing tools such as Woflan can already verify the soundness property for complex workflows encountered in practice. Moreover, if only free-choice WF-nets are allowed, as is the case in most of the workflow management systems, all requirements can be verified in polynomial time. See [14, 15] and [2] for more details.

4 Inheritance in the workflow-management domain

In this section, we discuss the usefulness of inheritance concepts in the context of workflow management. To address this issue, it is worthwhile to consider the following two trends:

- The shift from a “Sellers’ Market” to a “Buyers’ Market” in the last 30 years has resulted in an increase in the number of products and services offered to the customer. Consider for example mortgage loans; today, most financial institutions offer various types of mortgage loans. Moreover, the customer expects flexibility, i.e., the standard product or service has to be customized.
- Today’s enterprises have a complex and rapidly changing structure. Communication mechanisms such as Electronic Data Interchange (EDI) and the Internet have enabled Electronic commerce (E-commerce) and extended/virtual enterprises. As a result, many business processes have become interorganizational or intraorganizational.

The impact of these two trends on workflow management is significant. As a result of the first trend, the number of workflow processes and variants for these processes has increased considerably. The second trend has resulted in inter/intraorganizational workflows distributed over several sites and involving heterogeneous resources. Workflow processes are moving from long-lasting, well-defined, centralized business processes to dynamically changing, distributed business processes with many variants. Given these developments and the associated problems, inheritance concepts are of particular relevance for the next generation of workflow management systems. To illustrate this, we give a number of situations where the inheritance concept can be used to tackle certain problems.

4.1 Ad-hoc change

It is widely recognized that workflow management systems should provide flexibility. However, as indicated in the introduction, today's workflow management systems have problems dealing with change. A particular kind of change is *ad-hoc change*. Ad-hoc change affects individual cases, i.e., it refers to changes on a case-by-case basis rather than structural modifications of the workflow process definition. An ad-hoc change is typically the result of an error, a rare event, or special demands of the customer. Exceptions often result in ad-hoc changes. A typical example of ad-hoc change is skipping a task in case of an emergency. This kind of change is often initiated by some external factor. A typical dilemma related to ad-hoc change is the problem to decide what kinds of changes are allowed and the fact that it is impossible to foresee all possible changes.

The inheritance concepts presented in this paper can offer some support for ad-hoc change. The predefined workflow process definition is the superclass. The modified workflow process definition resulting from an ad-hoc change should be a subclass of this superclass under one of the four inheritance relations. By enforcing this requirement, certain properties are preserved. In a process resulting from an ad-hoc change that is a subclass of the predefined workflow process under protocol inheritance, new alternatives are offered but every sequence of tasks possible in the superclass is also possible in the subclass. For example, under protocol inheritance, it is possible to skip existing tasks by introducing "bypass" tasks. When projection inheritance is used, it is not allowed to skip existing tasks. However, it is possible to add new tasks in-between or in parallel. Which notion of inheritance is most appropriate depends on the situation. It is also possible to use different inheritance notions for different parts of the workflow process, e.g., subflows without interaction outside the company may be changed under life-cycle inheritance and subflows which communicate with external actors can only be changed under projection inheritance. Note that in general it is not possible to foresee all potential changes. The inheritance relations allow for formulating rules with respect to change rather than enumerating all possible exceptions.

Ad-hoc change typically leads to many variants of a given workflow process. Since such changes often correspond to exceptions, it is not desirable to combine all these variants in a single complex workflow. By using inheritance rather than creating a copy and modifying it each time a change is needed, only changes need to be stored. Moreover, as shown in Section 6, it is possible to provide aggregate management information.

4.2 Evolutionary change

New technology, new laws, and new market requirements lead to modifications of the workflow process definitions at hand. *Evolutionary change* refers to changes of a structural nature: From a certain moment in time, the process changes for all new cases to arrive at the system. This type of change is the result of a new business strategy, reengineering efforts, or a permanent alteration of external conditions (e.g., a change of law). Evolutionary change is initiated by the management to improve efficiency or responsiveness, or is forced by legislature or changing market demands. Evolutionary change always affects new cases but it may also influence old cases. Basically, there are four ways to deal with existing cases:

- *Restart*
  All existing cases are aborted and restarted in the new process. At any time, all cases use the same routing definition. For most workflow applications, it is not acceptable to restart cases because it is not possible to rollback work or it is too expensive to flush cases.

- *Abort*
  All existing cases are stopped and are not processed any further, i.e., all pending cases are aborted and considered to be ready. This approach is used if all existing cases are completed by hand. In general, this solution is not acceptable.

- *Proceed*
  Each case refers to a specific version of the workflow process. Newer versions do not affect old cases. Most workflow management systems support such a versioning mechanism. A drawback of this approach is that old cases cannot benefit from an improved routing definition. In addition, it might be a complex task to manage too many versions.

- *Transfer*
  Existing cases are transferred to the new process, i.e., they can directly benefit from evolutionary changes. Often, the transfer of cases is the preferred solution. The term dynamic change, introduced in the introduction, refers to the problem of transferring cases to a consistent state in the new process.

Evolutionary change can cause problems internal to the company such as the dynamic-change problem illustrated in Figure 1.1. In addition, it can also cause confusion for the outside world. If a customer is used to receiving goods before the bill and now starts receiving the bill before the goods, this may result in irritation. Therefore, change needs to be restricted and the designer needs to be aware of the fact that the environment can be affected by certain changes. The inheritance rules presented in the previous section can be used to restrict changes. Since observable behavior is one of the cornerstones of the defined inheritance notions (see Sections 3.1 and 2.3), the degree of change as observed by the environment can be quantified. If external business partners are involved in a change which does not satisfy certain inheritance requirements, they need to agree on such a change because the change might have externally noticeable effects. As a consequence, business partners will be able to notice the differences and need to act accordingly.

In Section 5, we show that by restricting evolutionary change to the four inheritance rules presented in the previous section the dynamic change-problem can be avoided.

4.3 Workflow templates

Although workflow processes within different enterprises have common elements, they are typically designed from scratch. Also within large companies, it is often not possible to specify a workflow process definition once and replicate it across all parts of the company that are in need of such a process. Local differences have to be taken into account and prohibit the use of one uniform solution. As a result, workflow processes are typically designed from scratch and the "wheel" is re-invented every day. To avoid re-inventing the wheel, one can use *workflow templates*. A workflow template is a standard design of a common workflow process. An enterprise or a department within the enterprise can use such a workflow template as the starting point for the design of a new workflow process. The standard solution provided by the template is changed to accommodate specific needs. The use of templates allows the designers to reflect local differences (resulting from specific regulations, organizational structures, and other particularities) and still re-use the common parts.

The idea of using templates for workflow processes is not new. Malone et al. defined a large number of process templates in the so-called "process handbook" [45]. Moreover, today's Enterprise Resource Planning (ERP) systems such as SAP R/3 (SAP AG, Walldorf, Germany, [38]) and Baan-ERP (Baan Company, Barneveld, The Netherlands, [49]) offer hundreds of ready-made workflow templates (often named business models or reference models) that can be used as a starting point for configuring the system. These workflow templates are often based on "best business practices" and reflect the experiences of leading enterprises. Although the set of workflow templates offered by today's workflow management systems is still limited, it is clear that the use of templates will increase to avoid starting from scratch every time a new workflow has to be designed.

Current ERP and workflow management systems provide limited support for templates. A designer has to make a copy of a template and customize it to accommodate specific needs. This "copy and modify" approach is not very sophisticated; any change is allowed and changes of the template do not affect the workflow processes designed using the template. Instead of the "copy and modify" approach, one could also use an approach based on inheritance. By establishing an inheritance relationship between the customized workflow process and the corresponding template, it is possible to restrict change and certain changes of the template can be transferred to the customized process.

4.4 E-commerce

Traditional *Electronic commerce* (E-commerce), mainly using *Electronic Data Interchange* (EDI), is rapidly moving to the Internet. Moreover, E-commerce is moving from long-lasting well-defined business relationships to a more dynamic situation, where parties having no prior trading relationship engage in a common business process. Consequently, the operational boundaries between organizations have become fluid. As a result, it is difficult to separate interorganizational business processes from the intraorganizational ones. E-commerce has complicated the management of business processes. The processes are scattered over multiple organizations and are subject to frequent changes. There are many problems which need to be solved to enable the enactment of workflows crossing organizational borders. These problems are of a conceptual (e.g., how to design interorganizational workflows), a technical (e.g., how to exchange data over the internet), a financial (e.g., how to distribute the benefits), and/or a managerial (e.g., who is responsible) nature. Two interesting conceptual problems that may benefit from the inheritance concepts presented in this paper are the following.

- How to agree on a common workflow without having to know all the details of each others business processes?
- How to allow for local changes (e.g., one of the business partners involved optimizes its internal process) without the need for global coordination?

To solve the first problem one could design a simple common workflow where only the tasks which are relevant for all partners are specified. Then, the common workflow is partitioned over the business partners involved, i.e., the global workflow process is split into local parts. Each business partner extends/refines the local workflow process until it can be made operational. However, changing the local workflow can cause problems, e.g., swapping two tasks can lead to deadlocks of the shared workflow. To avoid such problems, three of the four inheritance-preserving transformation rules can be used (*PPS*, *PJS*, and *PJ3S*). If the changes of the local workflow preserve *projection inheritance*, the other business partners cannot detect any differences and therefore problems such as deadlocks and livelocks can be avoided. The notion of projection inheritance is appropriate because it only allows for changes having *internal* effects. Note that protocol inheritance and life-cycle inheritance are not suitable for this application. An alternative route in one of the local workflows may lead to a deadlock of the overall workflow process. The second problem can be solved by using the same mechanism (i.e., projection inheritance); if local change is restricted to the transformation rules preserving projection inheritance, then the other business partners cannot detect any differences and there is no need for a new agreement on the global protocols.

The examples given in this section illustrate that inheritance concepts can be used to support ad-hoc change, evolutionary change, workflow templates, and E-commerce. Clearly, this paper is just a starting point for augmenting workflow management systems with inheritance concepts. For example, all the examples given in this paper focus on the process perspective, i.e., only the control flow and routing aspects are considered. Inheritance is equally important for the other perspectives dealing with the organization, data, applications, and operations. However, since the process perspective is dominant in workflow management applications, we restrict ourselves to this perspective. In the remainder, we show that inheritance can truly assist in dealing with the problems identified in Section 1.

5 Dynamic change

The problem of dynamic change was introduced using Figure 1.1. If a sequential process is changed to a parallel one, there are no problems. However, if the degree of parallelism is reduced, there are states in the old process which do not correspond to states in the new process. The state with a token in both p_1 and p_4 (right-hand side of Figure 1.1) cannot be mapped onto a state in the sequential process (left-hand side). Putting a token in i, s_1, or s_2 will result in the double execution of task *send_bill*. Putting a token in s_2, s_3, or o will result in the skipping of (at least) task *send_goods*. The problem identified does not only apply to the situation where the degree of parallelism is changed. For example, swapping or removing tasks may lead to similar problems. This is the reason most workflow management systems do not allow dynamic change, i.e., if a workflow process is changed, then all existing cases are handled the old way and the new process only applies to new cases. Every case has a pointer to a version of the workflow and each version is maintained as long as there are cases pointing to it. For some applications, this solution will do. However, if the flow time of a case is long, it may be unacceptable to process running cases the old way. Consider for example the change of a 4-year curriculum at a university to a 5 year one. It is too expensive to offer both curricula for a long time. Sooner or later, cases (i.e., students) need to be transferred. Other examples are mortgage loans and insurance policy's with a typical flow time of decades. Maintaining old versions of a process is often too expensive and may cause managerial problems. It is also possible that there are regulations (e.g., new laws) enforcing a dynamic change.

The inheritance-preserving transformation rules do not *solve* the problem indicated in Figure 1.1. The only way to avoid the incorrect execution of cases is to postpone the transfer of running cases in state $[p_1, p_4]$. The inheritance-preserving transformation rules can be used to *avoid* such problems by restricting change to those changes where a correct transfer is always possible.

The remainder of this section is organized as follows. First, we introduce the notion of a transfer rule, i.e., a rule to map cases from one workflow process definition to another. Second, we give concrete, generic transfer rules to map cases from a superclass to a subclass (i.e., specialization). Third, we provide generic rules to support generalization, i.e., mapping cases from a subclass to a superclass. Fourth, we discuss related work on dynamic change. Finally, we explain how our approach can be combined with so-called change regions. This combined approach can cope with changes such as the one in Figure 1.1 that prohibit an immediate transfer of cases.

5.1 Valid transfer rules

In this subsection, we assume the presence of two workflow process definitions: the old one and the new one. Cases present in the old process at the moment of change need to be transferred from the old workflow process definition to the new one, i.e., each case in the old process definition has to be removed and mapped onto a new case in the new process definition. Since the state of the case in the new process definition depends on the state of the case in the old process definition, we need to define a *transfer rule*.

Definition 5.1. (Transfer rule) Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$ be two workflow process definitions in $\mathcal{W}$. A *transfer rule* r from N_0 to N_1 (notation $N_0 \overset{r}{\leadsto} N_1$) is a partial function mapping markings of N_0 onto markings of N_1, i.e., $r : \mathcal{B}(P_0) \not\rightarrow \mathcal{B}(P_1)$.

A transfer rule maps states of an old workflow process definition onto states of a new workflow process definition. Note that the function can be partial ($dom(r) \subset \mathcal{B}(P_0)$), i.e., if the state of the case is not in the domain of the transfer rule, then the case is not transferred. Clearly, not every transfer rule is acceptable. The transfer of a case according to a transfer rule should not result in deadlocks or other anomalies. In this paper, we consider a transfer rule to be acceptable if and only if every transfer of a case results in a state in the new workflow process definition which is also reachable by newly initiated cases. A transfer rule satisfying this property is called valid.

Definition 5.2. (Valid transfer rule) Let N_0 and N_1 be two workflow process definitions in $\mathcal{W}$ and r a transfer rule $N_0 \overset{r}{\leadsto} N_1$. Transfer rule r is *valid* if and only if, for all $s \in dom(r) \cap [N_0, [i]\rangle$, $r(s) \in [N_1, [i]\rangle$.

Basically, a valid transfer rule makes sure that the soundness property of the target workflow net is guaranteed for cases that are transferred from another workflow process definition.

Property 5.3. *Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$ be two workflow process definitions in $\mathcal{W}$ and r a valid transfer rule $N_0 \overset{r}{\leadsto} N_1$. For any $s \in dom(r) \cap [N_0, [i]\rangle$, the following conditions are satisfied:*

1. *safeness: for any place $p \in P_1$, $r(s)(p) \leq 1$,*
2. *proper completion: if $o \in r(s)$, then $r(s) = [o]$, and*
3. *absence of deadlock: $[o] \in [N_1, r(s)\rangle$.*

Proof. It follows directly from Definitions 2.24 (Workflow process definition), 2.22 (Soundness), and 5.2 (Valid transfer rule). □

There are three generic transfer rules which are guaranteed to be valid.

Definition 5.4. (r_i, r_o, and $r_\emptyset$) Let N_0 and N_1 be two workflow process definitions in $\mathcal{W}$. Transfer rules $N_0 \overset{r_i}{\leadsto} N_1$, $N_0 \overset{r_o}{\leadsto} N_1$, and $N_0 \overset{r_\emptyset}{\leadsto} N_1$ are defined as follows:

- r_i is the transfer rule which maps every possible marking onto $[i]$, i.e., for any $s \in [N_0, [i]\rangle$, $r_i(s) = [i]$.
- r_o is the transfer rule which maps every possible marking onto $[o]$, i.e., for any $s \in [N_0, [i]\rangle$, $r_o(s) = [o]$.

- $r_{\emptyset}$ is the transfer rule with the empty domain, i.e., $dom(r_{\emptyset}) = \emptyset$.

Property 5.5. *r_i, r_o, and $r_{\emptyset}$ are valid.*

Proof. It follows directly from Definitions 2.24 (Workflow process definition), 2.22 (Soundness), 5.2 (Valid transfer rule), and 5.4; states $[i]$ and $[o]$ are reachable in any sound workflow net and the transfer rule with the empty domain is trivially valid. $\Box$

Transfer rules r_i, r_o, and $r_{\emptyset}$ correspond to three of the four policies described in Section 4.2. Rule r_i corresponds to restarting all existing cases, r_o corresponds to aborting all existing cases, and $r_{\emptyset}$ corresponds to completing all running cases according to the old process definition (i.e., the use of a versioning mechanism). In the remainder of this section, we do not consider these trivial transfer rules; we focus on transfer rules which lead to a direct transfer (i.e., no postponement such as in $r_{\emptyset}$) to a meaningful state (i.e., not by default to $[i]$ or $[o]$). Each of the rules presented corresponds to one of the inheritance-preserving transformation rules presented in Section 3.2.

5.2 Transfer of cases from superclass to subclass

In this paper, we consider two types of transfer rules, namely from a class to a subclass and from a class to a superclass. In this subsection, we present transfer rules mapping states of a class onto states of a subclass. Each of the transfer rules is based on one of the inheritance-preserving transformation rules of Section 3.2. Before we introduce the transfer rules corresponding to respectively *PPS*, *PTS*, *PJS*, and *PJ3S*, we introduce the identity function as a generic transfer rule.

Definition 5.6. **(r_{id})** Let N_0 and N_1 be two workflow process definitions in $\mathcal{W}$. $N_0 \overset{r_{id}}{\rightsquigarrow} N_1$ is the transfer rule which corresponds to the identity function, i.e., for any $s \in [N_0, [i]\rangle$, $r_{id}(s) = s$.

Transfer rule r_{id} is not necessarily valid. A state in the first workflow process definition does not have to exist in the second process definition. However, r_{id} turns out to be a suitable transfer rule for *PPS*, *PTS*, and *PJS*.

The first transfer rule is based on the inheritance-preserving transformation rule illustrated by Figure 3.11 (rule *PPS*, see Theorem 3.12). Since *PPS* only adds new alternative behavior and does not restrict the behavior of the part corresponding to the "old" workflow process definition in any way, the case can simply be transferred without changing its state.

Theorem 5.7. **(Transfer rule r_{PPS})** *Let N_0 and N_1 be two workflow process definitions satisfying the requirements stated in Theorem 3.12. Mapping $r_{PPS} = r_{id}$ is a transfer rule $N_0 \overset{r_{PPS}}{\rightsquigarrow} N_1$ that is valid.*

Proof. Recall that *PPS* is a special case of *PTS*. Therefore, the validity of r_{PPS} follows directly from the validity of the transfer rule r_{PTS} which is presented next (see Theorem 5.9). However, it is easy to see that r_{PPS} is valid: The extension N only adds behavior, i.e., $[N_0, [i]\rangle \subseteq [N_1, [i]\rangle$. Therefore, the identity function is valid. $\Box$

Figure 5.8 illustrates transfer rule r_{PPS}. Inheritance-preserving transformation rule *PPS* has been used to extend the old workflow process definition on the left-hand-side with task *inform_customer* which can be executed at any point between registration and filing. Transfer rule r_{PPS} transfers each case from the left-hand-side process definition to the right-hand-side process definition without changing the state. Figure 5.8 also illustrates transfer rule r_{PPS}^{-1} which is defined in Section 5.3.

The second transfer rule is based on inheritance-preserving transformation rule *PTS* and is identical to r_{PPS}.

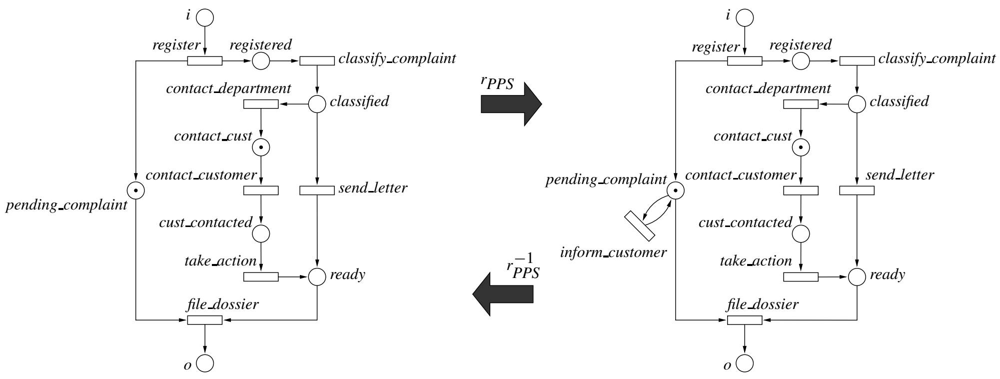

Figure 5.8: Transfer rule r_{PPS}.

Theorem 5.9. (Transfer rule r_{PTS}) *Let N_0 and N_1 be two workflow process definitions satisfying the requirements stated in Theorem 3.14. Mapping $r_{PTS} = r_{id}$ is a transfer rule $N_0 \overset{r_{PTS}}{\rightsquigarrow} N_1$ that is valid.*

Proof. Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$ be two workflow process definitions satisfying the requirements stated in Theorem 3.14. Since $P_0 \subseteq P_1$ and the extension N only enables new behavior (rather than restricting the existing behavior), it follows that $[N_0, [i]\rangle \subseteq [N_1, [i]\rangle$. This observation implies that the identity function r_{PTS} is valid. □

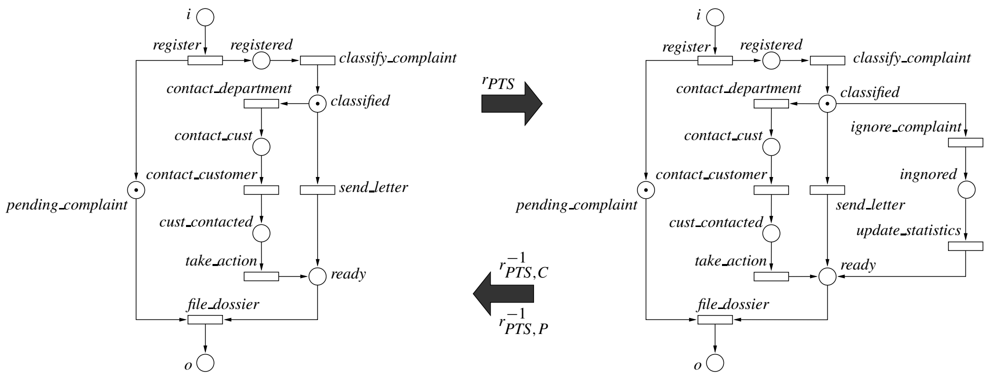

Figure 5.10: Transfer rule r_{PTS}.

Transfer rule r_{PTS} is illustrated in Figure 5.10. Using inheritance-preserving transformation rule *PTS*, the old workflow process definition on the left-hand-side is extended with an alternative branch containing tasks *ignore_complaint* and *update_statistics*. Every state in the left-hand-side process definition is also reachable in the right-hand-side process definition. Therefore, it is easy to see that the transfer rule r_{PTS} is valid in this particular situation. (Note that *update_statistics* is not present in Figure 3.6. The task has been added to the WF-net shown in Figure 5.10 to introduce a subclass which has states not present in the superclass, i.e., the states marking place *ignored*.)

The third transfer rule can be used when new tasks are inserted between existing sequential tasks as defined in Theorem 3.16 (i.e., transformation rule *PJS*).

Theorem 5.11. (Transfer rule r_{PJS}) *Let N_0 and N_1 be two workflow process definitions satisfying the requirements stated in Theorem 3.16. Mapping $r_{PJS} = r_{id}$ is a transfer rule $N_0 \overset{r_{PJS}}{\rightsquigarrow} N_1$ that is valid.*

Proof. The proof is similar to the proofs of Theorems 5.7 and 5.9. Since $P_0 \subseteq P_1$ and the added part N only inserts new behavior and does not restrict the existing behavior, it follows that $[N_0, [i]\rangle \subseteq [N_1, [i]\rangle$. This implies that the identity function r_{PJS} is valid. $\square$

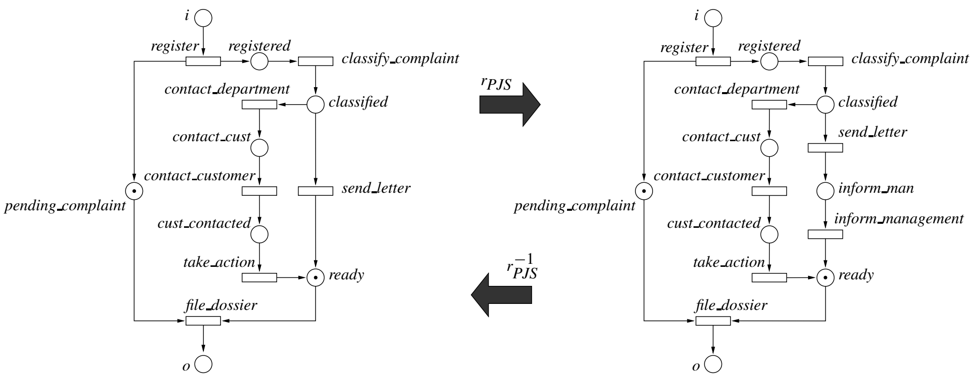

Figure 5.12: Transfer rule r_{PJS}.

Figure 5.12 illustrates transfer rule r_{PJS}. Inheritance-preserving transformation rule *PJS* has been used to insert task *inform_management* between *send_letter* and *ready*. The addition of this task only introduces new states. Therefore, transfer rule r_{PJS} transfers each case without changing the state. At a glance, an alternative transfer rule might be a mapping that transfers the token in place *ready* of the left-hand process definition to place *inform_man* of the right-hand process definition. Such a transfer rule would imply that the newly added task *inform_management* must be executed for the transferred case. However, such a transfer is only meaningful if the token in place *ready* of the left-hand process definition is the result of executing task *send_letter*. Clearly, this is not necessarily the case.

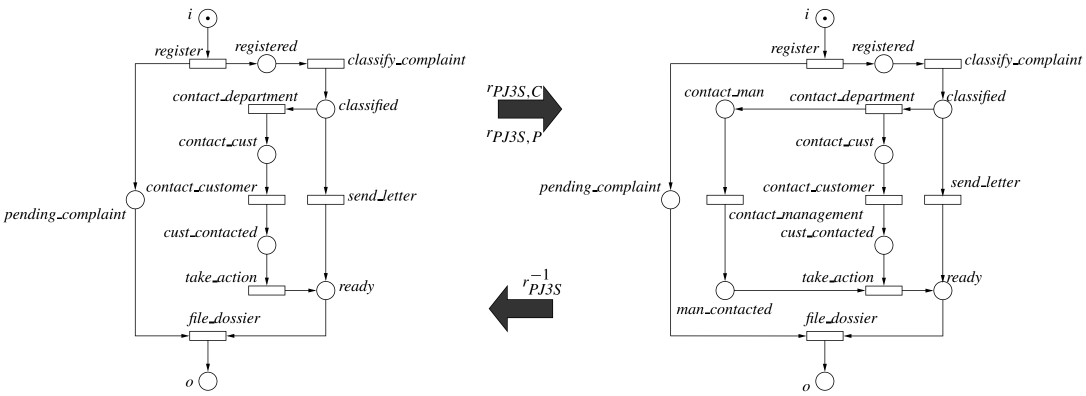

Figure 5.13: Transfer rules $r_{PJ3S,C}$ and $r_{PJ3S,P}$.

Transfer rules r_{PTS}, r_{PPS}, and r_{PJS} are rather trivial because additional behavior (i.e., alternative branches or parts inserted in-between existing parts) is introduced without eliminating existing states. The transfer of cases corresponding to transformation rule *PJ3S* is more complex because *PJ3S* adds parallel behavior rather than additional behavior. When adding parallel behavior, it may be necessary to mark places in the newly added parts. Consider for example the two workflow process definitions shown in Figure 5.13. The left-hand-side process definition has been transformed into the right-hand-side process definition using transformation rule *PJ3S*; task *contact_management* has been added such that it can be executed in parallel with *contact_customer*. Clearly, the state with a token in both *pending_complaint* and *registered* in the left-hand-side process definition should be mapped onto the identical state in the right-hand-side process definition. Also tokens marking *i*, *classified*, *ready*, and *o* should be transferred to the same place in the right-hand-side process definition. However, if one of the places *contact_cust* or *cust_contacted* is marked, then transferring the case to the identical state will result in a deadlock. Consider for example a case with tokens in both *pending_complaint* and *contact_cust* present in the left-hand-side process definition. If this state is transferred without modifications to the right-hand-side process definition, task *take_action* can never be executed, because place *man_contacted* will never get marked. The only way to solve this problem is to add an additional token to either place *contact_man* or *man_contacted*. In a conservative approach, *contact_man* is marked and the new task *contact_management* is required to be executed before task *take_action* is performed. In a progressive approach, *man_contacted* is marked and task *take_action* can be executed without executing the new task *contact_management*.

Figure 5.13 illustrates two complicating issues when transferring cases under the inheritance-preserving transformation rule *PJ3S*: (1) Sometimes (but not always) additional tokens need to be added, and (2) when adding tokens, there are sometimes multiple ways to add these tokens (e.g., conservative or progressive approach). Closely observing the requirements stated in Theorem 3.19 provides a solution for the first issue; the implicit place q acts as some kind of counter indicating whether the newly added part should be marked with additional tokens. The second issue is dealt with by providing two transfer rules: a conservative or pessimistic one ($r_{PJ3S,C}$) and a progressive or optimistic one ($r_{PJ3S,P}$). To define these two rules and prove their validity, we use the following lemma.

Lemma 5.14. *Let* $N = (P, T, F, \ell)$ *and* $N^q = (P_q, T_q, F_q, \ell_q)$ *be two workflow process definitions in* $\mathcal{W}$ *and* q *a place in* U *such that* $q \in P_q$, $P = P_q \backslash \{q\}$, $T = T_q$, $F = F_q \cap ((P \times T) \cup (T \times P))$, $\ell = \ell_q$, *and* q *is implicit in* $(N^q, [i])$. *For any reachable marking* $s \in [N, [i]\rangle$ *and firing sequences* $\sigma_1, \sigma_2 \in T^*$ *such that* $(N, [i])\ [\sigma_1\rangle\ (N, s)$ *and* $(N, [i])\ [\sigma_2\rangle\ (N, s)$, *there is a unique marking* $s' \in [N^q, [i]\rangle$ *such that, for all* $p \in P$, $s'(p) = s(p)$ *and* $(N^q, [i])\ [\sigma_1\rangle\ (N^q, s')$ *and* $(N^q, [i])\ [\sigma_2\rangle\ (N^q, s')$.

Proof. It is well known that any firing sequence $\sigma \in T^*$ enabled in $(N, [i])$ and resulting in s (i.e., $(N, [i])\ [\sigma\rangle\ (N, s)$) is enabled in $(N^q, [i])$ (i.e., $(N^q, [i])[\sigma\rangle$) (see Definition 3.17 (Implicit place) and [17, 22]). Assume s_1 and s_2 are the two markings in $[N^q, [i]\rangle$ such that $(N^q, [i])\ [\sigma_1\rangle\ (N^q, s_1)$ and $(N^q, [i])\ [\sigma_2\rangle\ (N^q, s_2)$. It follows from Definition 3.17 that, for all $p \in P$, $s_1(p) = s_2(p) = s(p)$. Thus, it remains to be proven that $s_1(q) = s_2(q)$. Assume $s_1(q) \neq s_2(q)$. Without loss of generality, we can assume that $s_1(q) > s_2(q)$. Since N is a workflow process definition, there is a σ_3 such that $(N, s)\,[\sigma_3\rangle\,(N, [o])$ (see Definition 2.24). Since q is implicit in $(N^q, [i])$, σ_3 is also enabled in (N^q, s_1) and (N^q, s_2). Let s_1' and s_2' be two markings such that $(N^q, s_1)\,[\sigma_3\rangle\,(N^q, s_1')$ and $(N^q, s_2)\,[\sigma_3\rangle\,(N^q, s_2')$. It follows from the fact that $s_1 > s_2 \geq s$ that $s_1' > s_2' \geq [o]$. However, the fact that $s_1' > [o]$ contradicts the fact that N^q is a workflow process definition (no proper completion; see Definition 2.22 (Soundness)). This contradiction shows that $s_1 = s_2$, which completes the proof. □

For any two workflow process definitions N and N^q satisfying the requirements of Lemma 5.14, the lemma states that any two firing sequences leading to the same marking in N also lead to the same marking in N^q and that, in addition, these two markings are identical with respect to places of N. Lemma 5.14 implies that the function in the following definition is well defined.

Definition 5.15. Let $N = (P, T, F, \ell)$, $N^q = (P_q, T_q, F_q, \ell_q)$, and q be defined as in Lemma 5.14. Function $id_q : [N, [i]\rangle \to [N^q, [i]\rangle$ is defined as follows. For any reachable marking $s \in [N, [i]\rangle$ and firing sequence $\sigma \in T^*$ such that $(N,[i])\,[\sigma\rangle\,(N,s)$, $id_q(s) = s'$ where s' is the unique marking in $[N^q,[i]\rangle$ defined by $(N^q,[i])\,[\sigma\rangle\,(N^q,s')$.

Consider the projection-inheritance-preserving transformation rule *PJ3S* of Theorem 3.19. To define the requirements of this transformation rule, the implicit place q was added to the original workflow N_0. Place q is a virtual place (i.e., it is not really present in one of the workflow process definitions N_0 or N_1) and has been added to make sure that every activation of the extension defined by N (i.e., a firing of transition t_i) is followed by a deactivation (i.e., a firing of transition t_o). Note that workflow process definitions N_0 and N_0^q of Theorem 3.19 satisfy the requirements of Lemma 5.14 and Definition 5.15. Function id_q as defined in Definition 5.15 counts the number of tokens in q. Since workflow process definitions are safe, for any reachable marking s of N_0, either $id_q(s) = s$ or $id_q(s) = s + [q]$. Recall the two complications concerning the transfer of cases under the inheritance-preserving transformation rule *PJ3S* identified above. If id_q indicates that q is not marked, then it suffices to transfer a case from the superclass N_0 to the subclass N_1 without changing its state. However, if id_q indicates that q is marked (i.e., q contains one token), then the newly added parallel branch of behavior needs to be marked when transferring a case. There are at least two ways to mark the parallel part: (1) The output places of t_i are marked (the conservative/pessimistic approach) or (2) the input places of t_o are marked (the progressive/optimistic approach). Therefore, we define two transfer rules based on *PJ3S*: $r_{PJ3S,C}$ and $r_{PJ3S,P}$

Theorem 5.16. (Transfer rules $r_{PJ3S,C}$ and $r_{PJ3S,P}$) *Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$ be two workflow process definitions satisfying the requirements stated in Theorem 3.19. Moreover, let q and N_0^q be as defined in Theorem 3.19 and $id_q : [N_0,[i]\rangle \rightarrow [N_0^q,[i]\rangle$ as defined in Definition 5.15. Finally, assume that $S_q = \{s \in [N_0,[i]\rangle \mid id_q(s)(q) > 0\}$.*

- *If $r_{PJ3S,C}$ is a transfer rule $N_0 \overset{r_{PJ3S,C}}{\rightsquigarrow} N_1$ such that, for all $s \in \mathcal{B}(P_0)\backslash S_q$, $r_{PJ3S,C}(s) = s$ and, for all $s \in S_q$, $r_{PJ3S,C}(s) = s + t_i\bullet\backslash P_0$, then $r_{PJ3S,C}$ is valid.*
- *If $r_{PJ3S,P}$ is a transfer rule $N_0 \overset{r_{PJ3S,P}}{\rightsquigarrow} N_1$ such that, for all $s \in \mathcal{B}(P_0)\backslash S_q$, $r_{PJ3S,P}(s) = s$ and, for all $s \in S_q$, $r_{PJ3S,P}(s) = s + \bullet t_o\backslash P_0$, then $r_{PJ3S,P}$ is valid.*

Proof. Let $s \in [N_0,[i]\rangle$ and $\sigma \in {T_0}^*$ be such that $(N_0,[i])\,[\sigma\rangle\,(N_0,s)$. We have to prove that $r_{PJ3S,C}(s) \in [N_1,[i]\rangle$ and $r_{PJ3S,P}(s) \in [N_1,[i]\rangle$.

Let t_i, t_o, and N be as defined in Theorem 3.19. In the firing sequence σ, t_i and t_o occur alternatingly, i.e., at any point in the sequence the number of times t_i has occurred is equal to the number of times t_o has occurred or t_i has occurred one time extra. This property is a direct result of the fact that q is implicit in $(N_0^q,[i])$ (which means that σ is enabled in $(N_0^q,[i])$) and that N_0^q is a workflow process definition (which means that q is safe). Because t_i and t_o occur alternatingly in σ, we distinguish two possibilities:

1. The number of times t_i occurs in σ (possibly zero) is equal to the number of times t_o occurs in σ, which means that $s \notin S_q$. Note that σ is not necessarily enabled in $(N_1,[i])$ when t_i occurs in σ. Let $\sigma_* \in (T\backslash\{t_i,t_o\})^*$ be such that $(N, t_i\bullet)\,[\sigma_*\rangle\,(N,\bullet t_o)$. Such a firing sequence exists, because $(N,[p])$ is live and safe (see Theorem 3.19). Let σ' be a modification of the firing sequence σ where immediately after every occurrence of t_i the sequence σ_* is inserted. Sequence σ' is enabled in $(N_1,[i])$ and results in state s. Since $(N_1,[i])\,[\sigma'\rangle\,(N_1,s)$ and $r_{PJ3S,C}(s) = r_{PJ3S,P}(s) = s$ ($s \notin S_q$), both $r_{PJ3S,C}(s)$ and $r_{PJ3S,P}(s)$ are elements of $[N_1,[i]\rangle$.

2. Transition t_i occurs precisely once more in sequence σ than transition t_o, which means that $s \in S_q$. Also in this case, σ is not necessarily enabled in $(N_1, [i])$ and in both states $r_{PJ3S,C}(s)$ and $r_{PJ3S,P}(s)$ some of the places in $P_1 \backslash P_0$ are marked. Construct sequence σ_* as before (i.e., $(N, t_i\bullet)\ [\sigma_*\rangle\ (N, \bullet t_o)$). Sequence σ' is the modification of the firing sequence σ where immediately before every occurrence of t_o the sequence σ_* is inserted. Sequence σ' is enabled in $(N_1, [i])$ and results in state $r_{PJ3S,C}(s) = s + t_i \bullet \backslash P_0$ (note that $s \in S_q$). Hence, $r_{PJ3S,C}(s) \in [N_1, [i]\rangle$. Sequence σ'' is the sequence obtained by concatenating σ' and σ_*. Also σ'' is enabled in $(N_1, [i])$ and results in state $r_{PJ3S,P}(s) = s + \bullet t_o \backslash P_0$. Hence, also $r_{PJ3S,P}(s) \in [N_1, [i]\rangle$.

□

In Figure 5.13, the left-hand-side process definition has been transformed into the right-hand-side process definition by adding task *contact_management* using transformation rule *PJ3S*. Consider a case in the left-hand process definition with a token in both *pending_complaint* and *contact_cust*. If transfer rule $r_{PJ3S,C}$ is used, this case is transferred to the state in the right-hand process definition where *pending_complaint*, *contact_cust*, and *contact_man* are marked. Transfer rule $r_{PJ3S,P}$ marks *man_contacted* instead of *contact_man*.

The transfer rules presented thus far map states of a superclass onto states of a subclass. If a series of inheritance-preserving transformation rules is applied to a workflow process definition, then it is possible to construct a composite transfer rule which is valid and maps any state of the original workflow process definition (superclass) onto the new process definition (subclass). The transfer rules presented in this subsection imply that for dynamically-changing workflow process definitions following the rules *PTS*, *PPS*, *PJS*, and *PJ3S* problems such as deadlocks, livelocks, and dangling references can be avoided. At the end of the next subsection, the construction of composite transfer rules is illustrated by means of our running example.

5.3 Transfer of cases from subclass to superclass

The transfer rules of the previous subsection assume that cases are transferred from a superclass to a subclass. However, one can think of applications where the inheritance-preserving transformation rules presented in Section 3.2 are applied in the *reverse* direction. Note that none of the rules of Section 3.2 assumes a direction. Therefore, the inheritance-preserving transformation rules can also be applied to create a superclass based on a subclass. This means that a workflow process definition is not extended but reduced (i.e., parts of the workflow process definition are removed). For example, a parallel branch can be removed by applying *PJ3S* in the reverse direction. If the inheritance-preserving transformation rules are applied in the reverse direction, we also need transfer rules which map states of the subclass onto states of the superclass. These transfer rules are presented in the remainder of this section.

First, let us consider a subclass workflow process definition and a superclass workflow process definition constructed by applying rule *PPS* of Theorem 3.12 in the reverse direction. Transfer rule r^{-1}_{PPS} is used to map states from the subclass to the superclass.

Theorem 5.17. (Transfer rule r^{-1}_{PPS}) *Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$ be two workflow process definitions satisfying the requirements stated in Theorem 3.12. Moreover, let p be the place as defined in Theorem 3.12 and let r^{-1}_{PPS} be a transfer rule $N_1 \overset{r^{-1}_{PPS}}{\rightsquigarrow} N_0$ such that, for all $s \in \mathcal{B}(P_0)$, $r^{-1}_{PPS}(s) = s$, and, for all $s \in \mathcal{B}(P_1) \backslash \mathcal{B}(P_0)$, $r^{-1}_{PPS}(s) = s \restriction P_0 + [p]$. Transfer rule r^{-1}_{PPS} is valid.*

Proof. Transformation rule *PPS* is a special case of rule *PTS* of Theorem 3.14. This can be proven by assuming that the places p_i and p_o of Theorem 3.14 are both equal to place p. Under this assumption, transfer rule r_{PPS}^{-1} equals the transfer rules corresponding to *PTS* applied in the reverse direction which are presented next. Both transfer rules $r_{PTS,C}^{-1}$ and $r_{PTS,P}^{-1}$ stated in Theorem 5.20 correspond to r_{PPS}^{-1}. Thus, the validity of r_{PPS}^{-1} follows immediately from the validity of $r_{PTS,C}^{-1}$ and $r_{PTS,P}^{-1}$. □

Figure 5.8 illustrates transfer rule r_{PPS}^{-1}. Inheritance-preserving transformation rule *PPS* has been used to reduce the old workflow process definition on the right-hand side into the new workflow process definition on the left-hand side. Since the removed task *inform_customer* did not add any new states r_{PPS}^{-1} corresponds (in this particular situation) to the identity function. If the removed part would have been a network of tasks, then all tokens in the removed part would have been mapped onto one single token in place *pending_complaint*.

Given a workflow process definition and a subclass of this process definition constructed by means of transformation rule *PPS*, the transfer of a case from the superclass to the subclass and back yields the original state.

Property 5.18. *Let* N_0, N_1, r_{PPS}, *and* r_{PPS}^{-1} *be as defined in Theorems 5.7 and 5.17. For any* $s \in [N_0, [i]\rangle$, $r_{PPS}^{-1}(r_{PPS}(s)) = s$.

Proof. The property follows immediately from the definitions of the transfer rules in Theorems 5.7 and 5.17. □

Generally, the converse does not hold. Transfer rule r_{PPS} corresponds to specialization and r_{PPS}^{-1} corresponds to generalization. If specialization follows generalization, it may not be possible to reconstruct the original state because information is lost during the generalization step.

Second, we consider the transfer of cases under the inheritance-preserving transformation rule *PTS* applied in the reverse direction. Consider the two workflow process definitions shown in Figure 5.10. The old process definition on the right-hand-side is reduced by removing the alternative branch containing tasks *ignore_complaint* and *update_statistics*. Note that we now have to transfer cases from "right to left" rather than from "left to right." For a case not in the alternative branch to be removed, the transfer is simple; the case can be transferred without changing its state. For a case in this alternative branch (i.e., places *pending_complaint* and *ignored* are marked), the transfer is more complicated since place *ignored* is not present in the new process definition. There are two ways to deal with this problem: Either the corresponding token can be moved to the point where the alternative branch starts (i.e., place *classified*; *conservative* approach) or it can be moved to the point where the alternative branch ends (i.e., place *ready*; *progressive* approach). Therefore, we give two valid transfer rules: $r_{PTS,C}^{-1}$ and $r_{PTS,P}^{-1}$.

Before we formulate $r_{PTS,C}^{-1}$ and $r_{PTS,P}^{-1}$, we need to consider another problem which is illustrated by Figure 5.19. (The net shown in Figure 5.19 without transition x is taken from [24].) Suppose we remove task x by applying rule *PTS* in the reverse direction, i.e., the old workflow process definition is the process definition with x and the new workflow process definition is the process definition without x. In the old process definition, the marking with a token in both p_3 and p_6 is reachable by firing t_1, t_2, and x. Although p_3 and p_6 are still present in the new workflow process definition, this marking is no longer reachable after removing x. This situation is rather exceptional; normally, the removal of an alternative branch of behavior such as the one modeled by transition x does not change the set of reachable markings with respect to the set of places that remain in the new workflow. If such a change does occur, it is not possible to come up with an elegant transfer rule which is valid. Recall

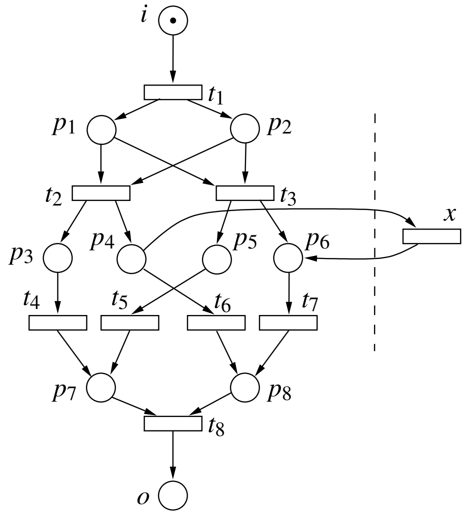

Figure 5.19: Transition x introduces a marking not reachable in the workflow process definition without x.

that a transfer rule is only valid if every transfer results in a state of the new WF-net that is also reachable from the initial marking in the new workflow process definition. Therefore, to define the transfer rules $r^{-1}_{PTS,C}$ and $r^{-1}_{PTS,P}$, we add the requirement that the removed part does not change the behavior in the remaining part. To formalize this requirement, we use the virtual transition x defined in Theorem 3.14. Transition x emulates the behavior of the removed part. Therefore, it is required that x does not influence the set of reachable markings.

Theorem 5.20. (Transfer rules $r^{-1}_{PTS,C}$ and $r^{-1}_{PTS,P}$) *Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$ be two workflow process definitions satisfying the requirements stated in Theorem 3.14. Moreover, let p_i, p_o, x, and N^x_0 be as defined in Theorem 3.14. Finally, assume that $[N^x_0, [i]\rangle = [N_0, [i]\rangle$.*

- *Let $r^{-1}_{PTS,C}$ be a transfer rule $N_1 \overset{r^{-1}_{PTS,C}}{\rightsquigarrow} N_0$ such that, for all $s \in \mathcal{B}(P_0)$, $r^{-1}_{PTS,C}(s) = s$, and, for all $s \in \mathcal{B}(P_1)\backslash\mathcal{B}(P_0)$, $r^{-1}_{PTS,C}(s) = s \upharpoonright P_0 + [p_i]$. Transfer rule $r^{-1}_{PTS,C}$ is valid.*

- *Let $r^{-1}_{PTS,P}$ be a transfer rule $N_1 \overset{r^{-1}_{PTS,P}}{\rightsquigarrow} N_0$ such that, for all $s \in \mathcal{B}(P_0)$, $r^{-1}_{PTS,P}(s) = s$, and, for all $s \in \mathcal{B}(P_1)\backslash\mathcal{B}(P_0)$, $r^{-1}_{PTS,P}(s) = s \upharpoonright P_0 + [p_o]$. Transfer rule $r^{-1}_{PTS,P}$ is valid.*

Proof. Let $s \in [N_1, [i]\rangle$ and $\sigma \in T_1{}^*$ be such that $(N_1, [i])\ [\sigma\rangle\ (N_1, s)$. We have to prove that $r^{-1}_{PTS,C}(s) \in [N_0, [i]\rangle$ and $r^{-1}_{PTS,P}(s) \in [N_0, [i]\rangle$. Let N be the labeled P/T net as defined in Theorem 3.14. We show that there are firing sequences $\sigma', \sigma'' \in (T_0 \cup \{x\})^*$ such that $(N^x_0, [i])\ [\sigma'\rangle\ (N^x_0, r_{PTS,C}(s))$ and $(N^x_0, [i])\ [\sigma''\rangle\ (N^x_0, r_{PTS,P}(s))$. Since $[N^x_0, [i]\rangle = [N_0, [i]\rangle$, this result suffices to prove that $r^{-1}_{PTS,C}$ and $r^{-1}_{PTS,P}$ are valid.

Let $T^I = \{t \in T \mid \bullet t = \{p_i\}\}$ and $T^O = \{t \in T \mid t\bullet = \{p_o\}\}$. Moreover, let y and N be as defined in Theorem 3.14. In the firing sequence σ, transitions in $T^I \cap T^O$ have the same effect in $(N_1, [i])$ as transition x in $(N^x_0, [i])$. Let $\sigma_x \in (T_1 \cup \{x\})^*$ be the sequence σ with each transition in $T^I \cap T^O$ replaced by transition x. As a result, in sequence σ_x, transitions in T^I and T^O occur alternatingly, i.e., at any point in the sequence the number of times that a transition in T^I has occurred is equal to the number of times that a transition in T^O has occurred or a transition in T^I has occurred one time extra. This property follows from the fact that $(N_1, [i])$ is safe and that $(N, [p_i])$ is live and safe (see Theorem 3.14). Because the transitions in T^I and T^O occur alternatingly in σ_x, we distinguish two possibilities:

1. Assume that the number of times that a transition in T^I occurs in σ_x is equal to the number of times that a transition in T^O occurs in σ_x. Since $(N, [p_i])$ is live and safe, firing a transition in T^O removes all tokens in $P\backslash\{p_i, p_o\}$. Since, in $(N_1, [i])$, the only way to mark places in $P\backslash\{p_i, p_o\}$ is to fire transitions in T^I, it follows that $s \in \mathcal{B}(P_0)$. Consider all subsequences of σ_x that start with a transition in T^I, end with an occurrence of a transition in T^O, and contain no other occurrences of transitions in T^I or T^O. For each such a subsequence, replace all occurrences of transitions in T by a single occurrence of transition x at some arbitrary position among the transitions remaining in the subsequence. Let $\sigma'_x \in (T_0 \cup \{x\})^*$ be the resulting firing sequence. Clearly, σ'_x is a sequence enabled in $(N_0^x, [i])$ and $(N_0^x, [i])\ [\sigma'_x\rangle\ (N_0^x, s)$ (i.e., $s \in [N_0^x, [i]\rangle$). Since $s \in \mathcal{B}(P_0)$ and, thus, $r^{-1}_{PTS,C}(s) = r^{-1}_{PTS,P}(s) = s$, this completes the proof in this case.

2. Assume that the number of times that a transition in T^I occurs in σ_x exceeds the number of times that a transition in T^O occurs in σ_x by one. Note that $P\backslash\{p_i, p_o\}$ cannot be empty in this case, because then T^I would equal T^O, which contradicts the assumption. It follows that $s \in \mathcal{B}(P_1)\backslash\mathcal{B}(P_0)$. Again, consider all subsequences of σ_x that start with an occurrence of a transition in T^I, end with an occurrence of a transition in T^O, and contain no other occurrences of transitions in T^I or T^O. For each such a subsequence, replace all occurrences of transitions in T by a single occurrence of transition x. The remaining occurrences of transitions of T in σ_x are simply removed. Let $\sigma'_x \in (T_0 \cup \{x\})^*$ be the resulting firing sequence. Sequence σ'_x is enabled in $(N_0^x, [i])$ and $(N_0^x, [i])\ [\sigma'_x\rangle\ (N_0^x, s \restriction P_0 + [p_i])$, because transition x emulates in $(N_0^x, [i])$ the behavior of subnet N in $(N_1, [i])$. Since $s \in \mathcal{B}(P_1)\backslash\mathcal{B}(P_0)$ and, thus, $r^{-1}_{PTS,C}(s) = s \restriction P_0 + [p_i]$, this completes the proof for $r^{-1}_{PTS,C}(s)$. Let sequence $\sigma''_x \in (T_0 \cup \{x\})^*$ be the sequence obtained by concatenating σ'_x and transition x. Also sequence σ''_x is enabled in $(N_0^x, [i])$ and results in state $r^{-1}_{PTS,P}(s) = s \restriction P_0 + [p_o]$, which completes the proof also for $r^{-1}_{PTS,P}(s)$.

□

The requirement in Theorem 5.20 that $[N_0^x, [i]\rangle = [N_0, [i]\rangle$ is essential for the validity of both transfer rules. Note that checking this requirement can be quite complex. However, from a practical point of view, it does not create a new problem. If a coverability graph is used to decide whether N_0^x is sound, then the requirement $[N_0^x, [i]\rangle = [N_0, [i]\rangle$ can be checked at no extra costs. First, construct a coverability graph for $(N_0, [i])$ and, then, add the arcs corresponding to x. If no new states are introduced, the requirement holds. In the remainder of this paper, we assume that the application of *PTS* is restricted to situations where the added part does not add any new behavior in the original part (i.e., $[N_0^x, [i]\rangle = [N_0, [i]\rangle$).

Consider Figure 5.10 where the old workflow process definition on the right-hand side is reduced by removing the alternative branch containing tasks *ignore_complaint* and *update_statistics*. If rule $r^{-1}_{PTS,C}$ is applied to a case which marks places *pending_complaint* and *ignored*, then the transfer results in the state which marks places *pending_complaint* and *classified*. Rule $r^{-1}_{PTS,P}$ maps the same case onto the state which marks *pending_complaint* and *ready*. For all other states, both $r^{-1}_{PTS,C}$ and $r^{-1}_{PTS,P}$ correspond to the identity function.

The following property states that transferring a case from a superclass to a subclass and back yields the original state.

Property 5.21. *Let N_0, N_1, r_{PTS}, $r^{-1}_{PTS,C}$, and $r^{-1}_{PTS,P}$ be as defined in Theorems 5.9 and 5.20. For any $s \in [N_0, [i]\rangle$, $r^{-1}_{PTS,C}(r_{PTS}(s)) = r^{-1}_{PTS,P}(r_{PTS}(s)) = s$.*

Proof. The desired result follows immediately from the definitions of the transfer rules in Theorems 5.9 and 5.20. □

Note that the property does not hold in the opposite direction, i.e., there may be states $s \in [N_1, [i]\rangle$ such that $r_{PTS}(r^{-1}_{PTS,C}(s)) \neq s$ and $r_{PTS}(r^{-1}_{PTS,P}(s)) \neq s$. Consider for example Figure 5.10, where the left-hand process definition corresponds to N_0 in Property 5.21 and the right-hand process definition corresponds to N_1. If $s \in [N_1, [i]\rangle$ equals [*pending_complaint*, *ignored*], then $r_{PTS}(r^{-1}_{PTS,C}(s))$ = [*pending_complaint*, *classified*] and $r_{PTS}(r^{-1}_{PTS,P}(s))$ = [*pending_complaint*, *ready*]. This example shows that transferring a case from a subclass to a superclass and back does not necessarily yield the original state.

Rule r^{-1}_{PJS} is a valid transfer rule when inheritance-preserving transformation rule *PJS* of Theorem 3.16 is used in the reverse direction. Recall that *PJS* inserts new tasks between a transition t_p and a place p. Rule r^{-1}_{PJS} removes tokens present in the inserted part and marks place p instead.

Theorem 5.22. (Transfer rule r^{-1}_{PJS}) *Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$ be two workflow process definitions satisfying the requirements stated in Theorem 3.16. Moreover, let p be the special place defined in Theorem 3.16 and let r^{-1}_{PJS} be a transfer rule $N_1 \overset{r^{-1}_{PJS}}{\rightsquigarrow} N_0$ such that, for all $s \in \mathcal{B}(P_0)$, $r^{-1}_{PJS}(s) = s$, and, for all $s \in \mathcal{B}(P_1) \backslash \mathcal{B}(P_0)$, $r^{-1}_{PJS}(s) = s \upharpoonright P_0 + [p]$. Transfer rule r^{-1}_{PJS} is valid.*

Proof. Let $T^O = \bullet p \cap T$. Moreover, let t_p be the special transition defined in Theorem 3.16. If transition t_p is an element of T^O, then the theorem reduces to Theorem 5.17, because in this case transformation rule *PJS* reduces to transformation rule *PPS* of Theorem 3.12. If t_p is not an element of T^O, then the proof is similar to the proof of Theorem 5.20 and is mainly based on the following observation. In the firing sequence σ, transition t_p and the transitions in T^O occur alternatingly. This property follows from the fact that $(N_1, [i])$ is safe and that $(N, [p])$ is live and safe. □

Figure 5.12 is used to illustrate transfer rule r^{-1}_{PJS}. If place *inform_man* in the old process definition on the right-hand side is not marked, a case can be transferred without changing its state. If place *inform_man* is marked, then the token in *inform_man* is moved to *ready*.

Property 5.23. *Let N_0, N_1, r_{PJS}, and r^{-1}_{PJS} be as defined in Theorems 5.11 and 5.22. For any $s \in [N_0, [i]\rangle$, $r^{-1}_{PJS}(r_{PJS}(s)) = s$.*

Proof. The property follows directly from Theorems 5.11 and 5.22. □

Rule r^{-1}_{PJ3S} is the remaining transfer rule which can be used to map states to a new workflow process definition where a parallel branch is removed (i.e., rule *PJ3S* applied in the reverse direction). This transfer rule simply removes all tokens in the deleted parallel part.

Theorem 5.24. (Transfer rule r^{-1}_{PJ3S}) *Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$ be two workflow process definitions satisfying the requirements stated in Theorem 3.19. Moreover, let r^{-1}_{PJ3S} be a transfer rule $N_1 \overset{r^{-1}_{PJ3S}}{\rightsquigarrow} N_0$ such that, for all $s \in \mathcal{B}(P_1)$, $r^{-1}_{PJ3S}(s) = s \upharpoonright P_0$. Transfer rule r^{-1}_{PJ3S} is valid.*

Proof. Let $s \in [N_1, [i]\rangle$ and $\sigma \in T_1{}^*$ be such that $(N_1, [i])\ [\sigma\rangle\ (N_1, s)$. We have to prove that $r^{-1}_{PJ3S}(s) \in [N_0, [i]\rangle$.

Let N be the labeled P/T net defined in Theorem 3.19. Let $\sigma' \in T_0{}^*$ be the firing sequence obtained by removing all occurrences of transitions in $T\backslash\{t_i, t_o\}$ from σ. Sequence σ' is enabled in $(N_0, [i])$. It is not difficult to verify that the state resulting from firing σ' in $(N_0, [i])$ is equal to s with respect to the places in P_0, i.e., $r^{-1}_{PJ3S}(s) = s \restriction P_0 \in [N_0, [i]\rangle$. □

In Figure 5.13, the right-hand-side process definition has been transformed into the left-hand-side process definition by removing task *contact_management*. Transfer rule r^{-1}_{PJ3S} removes any token in *contact_man* or *man_contacted* (if present) such that the case can be transferred from the subclass to the superclass.

Property 5.25. *Let N_0, N_1, $r_{PJ3S,C}$, $r_{PJ3S,P}$, and r^{-1}_{PJ3S} be as defined in Theorems 5.16 and 5.24. For any $s \in [N_0, [i]\rangle$, $r^{-1}_{PJ3S}(r_{PJ3S,C}(s)) = r^{-1}_{PJ3S}(r_{PJ3S,P}(s)) = s$.*

Proof. It follows directly from Theorems 5.16 and 5.24. □

For each inheritance-preserving transformation rule of Section 3.2, we have defined valid transfer rules for moving a case from a superclass to a subclass and vice versa. These rules are such that if a case is moved from the superclass to the subclass and back, the original state is obtained. (Generally, the converse does not hold.)

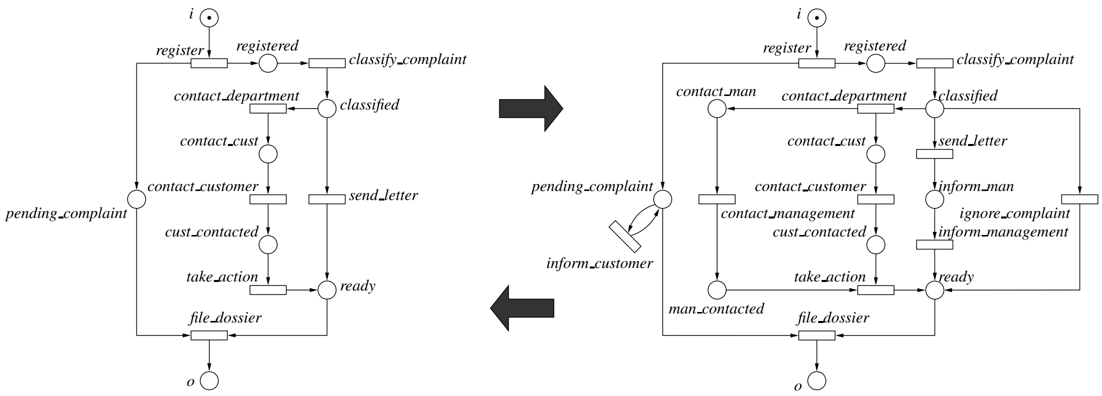

Figure 5.26: Dynamic change between a superclass (left-hand-side workflow process definition) and a subclass (right-hand-side process definition).

If a series of inheritance-preserving transformation rules is applied, the composition of the appropriate corresponding transfer rules yields a composite transfer rule which is valid. Consider for example the two workflow process definitions shown in Figure 5.26. The right-hand-side process definition is a subclass of the left-hand-side process definition under life-cycle inheritance and can be obtained by applying rules *PTS*, *PPS*, *PJS*, and *PJ3S*. Let r_C (r_P) be the transfer rule obtained by composing the transfer rules r_{PTS}, r_{PPS}, r_{PJS}, and $r_{PJ3S,C}$ ($r_{PJ3S,P}$). Consider a case in the superclass which marks *pending_complaint* and *cust_contacted*. If r_C is used to transfer this case to the subclass, the places *pending_complaint*, *cust_contacted*, and *contact_man* are marked after the transfer. If r_P is used, *man_contacted* is marked instead of *contact_man*. Let r_C^{-1} (r_P^{-1}) be the transfer rule obtained by composing the transfer rules $r^{-1}_{PTS,C}$ ($r^{-1}_{PTS,P}$), r^{-1}_{PPS}, r^{-1}_{PJS}, and r^{-1}_{PJ3S}. (Note that for this particular pair of workflow process definitions r_C^{-1} is equivalent to r_P^{-1}.) Consider a case in the subclass marking *pending_complaint* and *inform_man*. If r_C^{-1} is used to transfer this case to the superclass, the places *pending_complaint* and *ready* are marked afterwards.

To conclude, let us return to the problem illustrated by Figure 1.1. Has this problem (i.e., the dynamic-change bug) been solved? One might argue that the problem has not been solved, because the inheritance concepts do not provide a solution for this particular example. However, as explained in the beginning of this section, there is no satisfactory solution but to postpone the transfer (e.g., transfer rule $r_\emptyset$). The state with a token in both p_1 and p_4 (right-hand side of Figure 1.1) cannot be mapped onto a reasonable state in the sequential process definition (left-hand side). Putting a token in i, s_1, or s_2 will result in the double execution of task *send_bill*. Putting a token in s_2, s_3, or o will result in the skipping of task *send_goods*. The transfer rules presented in this section show that, if one restricts change to the inheritance-preserving transformation rules presented in Section 3.2, it is always possible to find a satisfactory transfer rule which is valid and, thus, the dynamic-change problem can be avoided. We have not formalized the term "satisfactory" but it is easy to see that the transfer rules do not lead to the unnecessary skipping or multiple execution of tasks. The transfer rules offer reasonable solutions for the types of changes typically used to adapt a workflow process definition: adding/removing alternative branches (*PTS*), adding/removing loops (*PPS*), adding/removing subflows between sequential tasks (*PJS*), and adding/removing parallel branches (*PJ3S*).

5.4 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. [18]) 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 1.1, 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 [26] 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 [27], 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 [25, 41], 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 [10] 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 1.1. The approach presented in [10] 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 [23]. Casati, Ceri, and Pernici [20] 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 [52] 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 [60, 61] 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. The reader interested in workflow change and Petri nets is also referred to [6] which contains several papers of the authors mentioned above. We also refer to the PhD thesis of Keddara [41] for a more complete overview of related work on dynamic change.

None of the work mentioned above uses an approach based on inheritance. The transfer rules based on the four inheritance-preserving transformation rules guarantee the preservation of the soundness property after a transfer. Moreover, 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.

5.5 Combining an approach based on inheritance with change regions

The dynamic-change bug illustrated by Figure 1.1 cannot be solved using an approach based on inheritance. If, for example, a case in the parallel process (right) needs to be migrated to the sequential process (left), then the inheritance-preserving transformation rules and other techniques presented in this paper are not of any assistance. There is not an acceptable way to migrate a case in state $[p_1, p_4]$ to the sequential process. The only way to avoid anomalies is to postpone the transfer of this case until *send_goods* is executed. The inheritance notions can only be used to avoid such a situation: If change is limited to the inheritance-preserving transformation rules, then it is always possible to circumvent the dynamic-change bug. In Section 4, it has been motivated that there are many situations where it is reasonable to limit change to one of the four inheritance notions. However, it is not realistic to expect that any change can be restricted in this way. As Figure 1.1 shows, there are situations where it makes sense to change the degree of parallelism. Moreover, there may be other situations where it makes sense to deliberately change to ordering of tasks. For these situations, none of four inheritance notions applies. Therefore, we propose an approach combining the the techniques presented in this paper with the techniques presented in [10, 23, 26, 27].

Suppose that we want to change a workflow process definition from N_0 to N_1 in such a way that not all changes are captured by our inheritance-preserving transformation rules. The first step in the combined approach is the identification of those changes that are captured by the transformation rules. Applying the transformations yields an intermediate workflow process definition N_2; as shown in this section, all cases of N_0 can be transferred to N_2. The second step in the combined approach is to identify the remaining changes as change regions. Every change region is defined by a pair of workflow process definitions (N_2^c, N_1^c) such that N_2^c, the old region, is a subnet of N_2 and N_1^c, the new region, is a subnet of N_1. For simplicity, let us assume that there is only one change region (N_2^c, N_1^c). For this change region, we can use the techniques presented in [10, 23, 26, 27]. For example, if the degree of parallelism is increased in the change region, then we can apply Theorem 8.2 of [26]. This theorem states that it is possible to migrate cases when moving to a more parallel process (upsizing), i.e., there is an acceptable function for migrating cases from N_2^c to N_1^c. Consider a workflow process where the change region (N_2^c, N_1^c) is described by Figure 1.1: The left-hand side describes the old region and the right-hand side describes the new region. Theorem 8.2 states that for any state of the old region it is possible to migrate the case to the new region. Thus, in this example, all cases can be transferred from the original workflow process definition N_0 via the intermediate process definition N_2 to the new process definition N_1 by combining the techniques of this paper and those of [26]. As another example, assume that the degree of parallelism is reduced in change region (N_2^c, N_1^c) (i.e., downsizing). In this case, Theorem 8.1 of [26] can be applied. This theorem states that a so-called *Synthetic Cut-Over Change* (SCOC), which temporarily adds both N_2^c to N_1^c to the new workflow process definition, can be used to deal with the problem. Consider again Figure 1.1. If the right-hand side describes the old region N_2^c and the left-hand side describes the new region N_1^c, then both the old and the new region need to be added temporarily to the new workflow process definition. New cases are handled according to the new region and existing cases are handled according to the old region. Thus, in this case, a complete transfer of all cases is not possible.

The above examples show how our approach can be combined with the techniques of [26]. Similarly, we can apply the techniques presented in [10, 23, 27]. The scope of these techniques is limited to those changes that are not captured by the inheritance-based techniques presented in this paper.

6 Management information

The transfer rules defined in the previous section are used to migrate cases from one workflow process definition to another workflow process definition. If all cases are migrated to the most recent workflow process definition, it is easy to provide aggregate management information. For each place p, it is possible to count the number of cases marking p. By indicating these numbers in the WF-net representing the most recent workflow process definition, the manager obtains a condensed view of the work-in-progress. However, in many situations, there are multiple versions and/or variants of the same workflow process. For evolutionary change, the number of versions is typically limited. In fact, if all cases are transferred directly after a change, there is just one active version. However, if the proceed policy (transfers are postponed until the case is handled completely, i.e., transfer rule $r_\emptyset$ of Definition 5.4) is used or transfers are delayed, there are multiple active versions. As indicated in the introduction, there may be various reasons for using such policies (e.g., legal constraints, technical problems, or managerial considerations). If the average flow time of cases is long and evolutionary changes occur frequently, there can be dozens of versions. Ad-hoc change may result in an even larger number of variants of the same workflow process. In fact, it is possible to end up in the situation where the number of variants is of the same order of magnitude as the number of cases. In Section 4, it has been pointed out that the shift from a “Sellers’ Market” to a “Buyers’ Market” leads to an increase in the number of products and services offered. Consider for example the number of variants of a specific car model (combinations of colors, engines, options, accessories). The number of variants may in fact exceed the number of cars actually sold. Clearly, the result of this shift is an increasing number of variants of a given workflow process. Moreover, today’s customers expect flexibility which may lead to even more new ad-hoc variants. To manage a workflow process with different versions/variants, it is desirable to have an aggregated view of the work-in-progress. Therefore, as indicated in Section 1, it is of the utmost importance to supply a manager with tools to obtain a condensed but accurate view of the status of the cases in the workflow process at hand.

In this section, we show that the inheritance notions introduced in this paper facilitate the construction of aggregate management information. First, we introduce the notion of a *management-information net*. Second, we define the notions of a *Maximal/Greatest Common Divisor* (MCD/GCD) and a *Minimal/Least Common Multiple* (MCM/LCM) of a number of workflow process definitions. Third, we discuss the utilization of the four inheritance-preserving transformation rules of Section 3.2 for constructing aggregate management information. Finally, we discuss several approaches for obtaining aggregate management information in the various application areas identified in Section 4.

6.1 Management-information nets

To provide a manager with succinct information about work-in-progress, we need *one* diagram *summarizing* the states of all running cases in all versions and/or variants of a workflow process. That is, the diagram should not show the states of individual cases nor should it show all the versions/variants of a workflow process; the information should be at an aggregate level. Every version/variant of a workflow process is represented as a workflow process definition (i.e., a sound WF-net). Therefore, it is reasonable to use a workflow process definition to present aggregate management information. We use the term *Management-Information net* (MI-net) to refer to a workflow process definition which is used to visualize work-in-progress.

Definition 6.1. (MI-net) Let $N_0, N_1, \ldots, N_{n-1}$, where n is some natural number, be n workflow process definitions in $\mathcal{W}$, which are versions/variants of the same workflow process. In addition, let N be a workflow process definition in $\mathcal{W}$. Net N is a *Management-Information net* (MI-net) for $N_0, N_1, \ldots, N_{n-1}$ if and only if there is a *total*, *valid* transfer rule for each version/variant, i.e., for each k, with $0 \leq k < n$, there is a transfer rule r_k such that $N_k \overset{r_k}{\rightsquigarrow} N$, $[N_k, [i]\rangle \subseteq dom(r_k)$, and r_k is valid.

Given a number of versions and/or variants of a workflow process and an MI-net with accompanying transfer rules, the states of every running case can be mapped onto the MI-net using the appropriate transfer rule. Note that there is no actual transfer of cases; only information about the state of a case is mapped onto the MI-net. Since we want to collect information about *all* cases, the transfer rules in Definition 6.1 are required to be *total*. Furthermore, because management information must be as accurate as possible, it is required that the transfer rules are *valid*.

Note that Definition 6.1 is not very restrictive. In fact, the definition does not impose any restrictions on an MI-net other than the requirement that it is a workflow process definition; any sound WF-net can serve as an MI-net. In principle, we can map cases from any set of workflow process definitions onto an arbitrary MI-net by using one of the trivial transfer rules given in Definition 5.4 (i.e., r_i or r_o). Therefore, the quality of management information can be low. In general, it is not easy to find a meaningful MI-net and non-trivial valid transfer rules.

Consider for example the three workflow process definitions shown in Figure 6.2. These process definitions are variants of the same workflow process. These variants can be the result of ad-hoc changes. Process definition N_0 represents, for example, the normal process. Process definition N_1 yields a slightly changed workflow where the parallel branch containing task *check1* is added. Process definition N_2 is another ad-hoc variant where task *check2* is added in-between *handle* and *archive*.

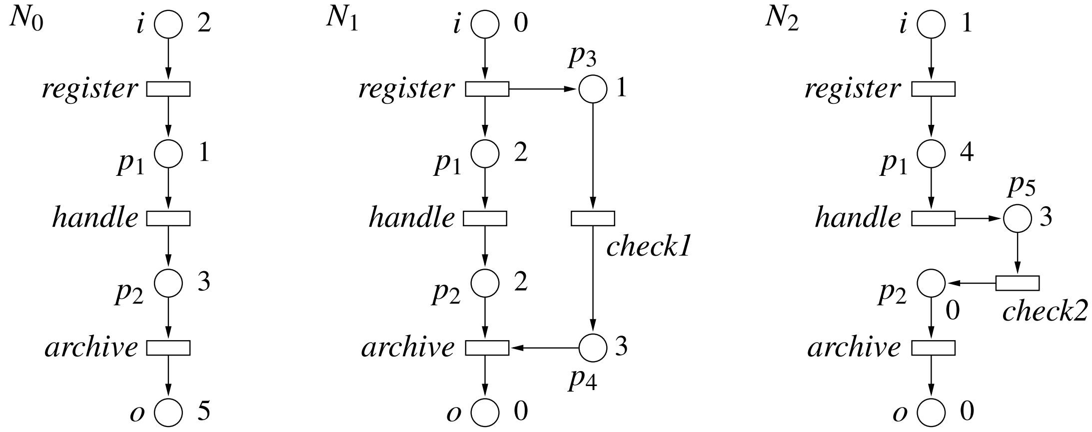

Figure 6.2: Three variants of a workflow process.

Note that the three workflow process definitions can also be the result of evolutionary changes (in which case they are called versions). Net N_0 is for example the oldest process definition, N_1 is the successor of N_0, and N_2 is the successor of N_1. Note that N_2 is not a subclass of N_1 under any of the four inheritance relations of Definition 3.4. This is not a problem since, for now, arbitrary changes are considered, i.e., also changes not respecting the inheritance relations. In the remainder of this section, we only use the term "variant" and not the term "version" when referring to an element of a set of workflow process definitions. However, the concepts are valid for both ad-hoc and evolutionary changes.

The numbers shown in Figure 6.2 are used to indicate the number of cases within a certain state for each variant. For example, in variant N_1, there are four cases. For each of these cases, *register* has been executed and for none of these cases *archive* has been executed. Task *handle* has been executed for two of these cases and *check1* has been executed for three of these cases. Note that, in total, there are 23 cases in the three variants shown in Figure 6.2: eleven in N_0, four in N_1, and eight in N_2. For these three simple variants, there is no real need for aggregate management information with respect to work-in-progress. However, one can imagine that, if the number of variants increases or the variants themselves become more complex, there is a need for aggregate information rather than separate status reports for each variant (as is shown in Figure 6.2). To accommodate this need, the cases have to be mapped onto an MI-net. Even for the three simple variants shown in Figure 6.2, it is not clear how to construct a meaningful MI-net. Should an MI-net be one of the variants? Should the MI-net emphasize the common parts as much as possible? Should it capture all possible routings? Another non-trivial question is how to obtain meaningful transfer rules.

Figure 6.3 shows two MI-nets for the three process variants shown in Figure 6.2. The left-hand-side MI-net N_{GCD} emphasizes the common part of the three variants. Each of the three variants contains the steps *register*, *handle*, and *archive* and these steps are always executed in this order. The right-hand-side MI-net N_{LCM} contains all tasks present in any of the variants in an effort to capture all possible routings. Note that the two MI-nets are augmented with numbers indicating the number of cases in each state. For example, in both MI-nets, place p_1 is labeled with $7 = 1 + 2 + 4$ indicating that seven cases in Figure 6.2 are in a state between *register* and *handle*, one in variant N_0, two in variant N_1, and four in variant N_2.

It is interesting to note that each of the three workflow process definitions shown in Figure 6.2 is a *subclass* under life-cycle inheritance of the left-hand MI-net N_{GCD}; N_0 is identical to N_{GCD}; N_1

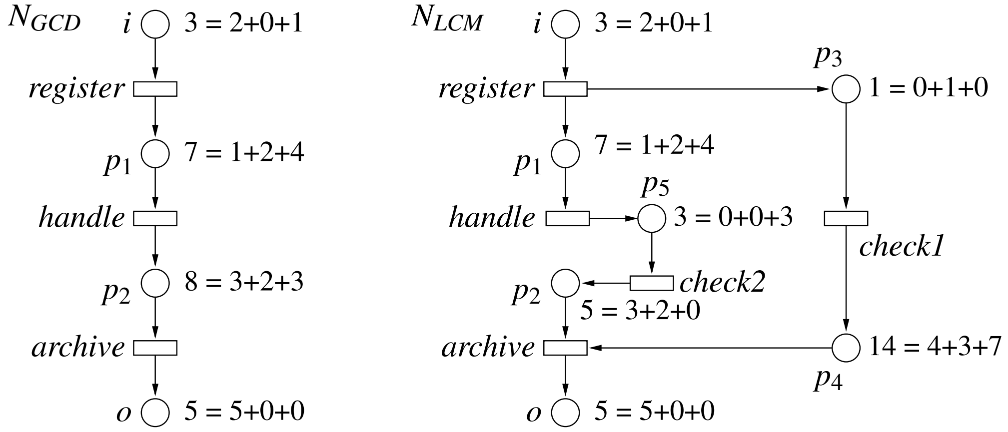

Figure 6.3: The Greatest Common Divisor (GCD) and Least Common Multiple (LCM) of the three workflow process definitions of Figure 6.2.

can be constructed from N_{GCD} by applying transformation rule *PJ3S* of Theorem 3.19 and N_2 can be constructed from N_{GCD} by applying rule *PJS* of Theorem 3.16. As a result, we can use the transfer rules r_{id}, r^{-1}_{PJ3S}, and r^{-1}_{PJS} of the previous section to map states of the three workflow process definitions of Figure 6.2 onto N_{GCD} of Figure 6.3. It is not difficult to verify that one obtains the numbers given in Figure 6.3 when doing so.

As mentioned, the other MI-net shown in Figure 6.3, N_{LCM}, attempts to capture all possible behaviors of the three workflow process definitions of Figure 6.2 rather than focusing on the common parts; by hiding *check1* and/or *check2*, one can find each of the three variants. This means that each of the three workflow process definitions shown in Figure 6.2 is a *superclass* of N_{LCM}: N_{LCM} can be constructed from N_0 by applying the inheritance-preserving transformation rules *PJS* and *PJ3S*; N_{LCM} can be constructed from N_1 by applying *PJS* and it can be obtained from N_2 by applying *PJ3S*. Consequently, we can use r_{PJS} of Theorem 5.11 to map states of N_1 onto N_{LCM}. Furthermore, states of N_2 can be mapped onto N_{LCM} using either $r_{PJ3S,C}$ or $r_{PJ3S,P}$ of Theorem 5.16 . States of N_0 can be mapped onto N_{LCM} using r_{PJS} and either $r_{PJ3S,C}$ or $r_{PJ3S,P}$. Note that, in Figures 6.2 and 6.3, the four cases in N_0 in-between *register* and *archive* as well as the seven cases in N_2 in-between *register* and *archive* have been mapped onto place p_4 of N_{LCM} using transformation rule $r_{PJ3S,P}$. As a result, place p_4 of N_{LCM} is labeled with $14 = 4 + 3 + 7$. (Place p_4 contains also information on the three cases residing in states marking p_4 in N_1.)

In the context of management information, there is a good reason to use transfer rule $r_{PJ3S,P}$ instead of rule $r_{PJ3S,C}$ if both rules are applicable. Technically, it is possible to use a conservative mapping based on $r_{PJ3S,C}$ (i.e., in the above example, twelve cases are mapped onto place p_3 in N_{LCM}) or a progressive mapping based on $r_{PJ3S,P}$ (i.e., just one case is mapped onto place p_3 in N_{LCM}). However, for the purpose of management information, it is not meaningful to use $r_{PJ3S,C}$. The conservative mapping results in a view which is too pessimistic. The cases in N_0 and N_2 in-between *register* and *archive* do not require the execution of task *check1*, whereas the conservative mapping of these cases onto N_{LCM} suggests that they do. Transformation rule $r_{PJ3S,C}$ does not provide an accurate estimate of work-in-progress. Therefore, we exclude this transformation rule for determining management information. In the remainder, we only use the other transfer rules of the previous section, namely r_{PTS}, r_{PPS}, r_{PJS}, $r_{PJ3S,P}$, $r^{-1}_{PTS,C}$, $r^{-1}_{PTS,P}$, r^{-1}_{PPS}, r^{-1}_{PJS}, and r^{-1}_{PJ3S}. Note that it is meaningful to consider both $r^{-1}_{PTS,C}$ and $r^{-1}_{PTS,P}$. If an alternative branch present in one variant of a workflow process is not present in the MI-net that is being used, cases for which part of the alternative branch has been executed can be moved to the point where the alternative branch starts (conservative view) or to the point where the alternative branch ends (progressive view). The question of which mapping is most accurate depends on the context.

The example of Figures 6.2 and 6.3 illustrates the basic idea of constructing aggregate management information. It shows that the inheritance concepts introduced in this paper can be useful in obtaining meaningful MI-nets and transfer rules. Clearly, the names N_{GCD} and N_{LCM} of the two MI-nets in Figure 6.3 are suggestive. In the next subsection, the notions of a *Maximal/Greatest Common Divisor* (MCD/GCD) of a number of variants of a workflow process and the notions of a *Minimal/Least Common Multiple* (MCM/LCM) of a number of variants are formalized. In Section 6.3, the role of the inheritance-preserving transformation rules of Section 3.2 and the transfer rules of Section 5 is studied in more detail.

6.2 Maximal common divisors and minimal common multiples of workflow process definitions

The left-hand-side WF-net N_{GCD} shown in Figure 6.3 is an MI-net which is a superclass of each of the three variants shown in Figure 6.2. As explained, the cases present in the three variants can be mapped onto N_{GCD} using respectively transfer rule r_{id} (see Definition 5.6), transfer rule r_{PJ3S}^{-1} (see Theorem 5.24), and transfer rule r_{PJS}^{-1} (see Theorem 5.22). One can think of N_{GCD} as the *intersection* or *greatest common divisor* (GCD) of the three process definitions shown in Figure 6.2; N_{GCD} contains the elements which are present in all variants. By coincidence, N_{GCD} equals one of the variants (N_0). The right-hand-side MI-net N_{LCM} shown in Figure 6.3 does not correspond to one of the variants. One can think of this MI-net as the *union* or *least common multiple* (LCM) of the three variants shown in Figure 6.2. However, the terms “intersection” and “union” may be misleading, because the straightforward intersection and union of the network structure of a set of workflow process definitions generally does not yield an MI-net. Therefore, we prefer the terms “GCD” and “LCM.” The notions of a GCD and an LCM are defined using the life-cycle-inheritance relation of Definition 3.4-4 and the auxiliary notions of a Maximal Common Divisor (MCD) and a Minimal Common Multiple (MCM). Recall that $\cong$, as defined in Definition 2.25, denotes behavioral equivalence of workflow process definitions.

Definition 6.4. (MCD/GCD, MCM/LCM) Let $N_0, N_1, \ldots, N_{n-1}$, where n is some natural number, and N be workflow process definitions in $\mathcal{W}$.

1. Net N is a *Maximal Common Divisor* (MCD) of $N_0, N_1, \ldots, N_{n-1}$ if and only if
   (a) $(\forall k : 0 \leq k < n : N_k \leq_{lc} N)$ and,
   (b) for any workflow process definition N' such that $(\forall k : 0 \leq k < n : N_k \leq_{lc} N')$ and $N' \leq_{lc} N$, $N' \cong N$.
2. Net N is a *Greatest Common Divisor* (GCD) of $N_0, N_1, \ldots, N_{n-1}$ if and only if, it is an MCD of $N_0, N_1, \ldots, N_{n-1}$ such that, for all MCDs N' of $N_0, N_1, \ldots, N_{n-1}$, $N' \cong N$.
3. Net N is a *Minimal Common Multiple* (MCM) of $N_0, N_1, \ldots, N_{n-1}$ if and only if
   (a) $(\forall k : 0 \leq k < n : N \leq_{lc} N_k)$ and,
   (b) for any workflow process definition N' such that $(\forall k : 0 \leq k < n : N' \leq_{lc} N_k)$ and $N \leq_{lc} N'$, $N' \cong N$.

4. Net N is a *Least Common Multiple* (LCM) of $N_0, N_1, \ldots, N_{n-1}$ if and only if, it is an MCM of $N_0, N_1, \ldots, N_{n-1}$ such that, for all MCMs N' of $N_0, N_1, \ldots, N_{n-1}$, $N' \cong N$.

Note that the notions of an MCD/GCD and an MCM/LCM are defined with respect to life-cycle inheritance and not with respect to the size of workflow process definitions (where the size of a workflow process definition is determined by its number of tasks). If N_{MCD} is an MCD of two workflow process definitions N_0 and N_1, then N_{MCD} typically contains *fewer* tasks than N_0 and N_1, which conforms to the intuitive notion of an MCD. On a first reading, the definition of an MCD of a number of process variants might be counterintuitive because an MCD is required to be a *super*class of the process variants. Similarly, if N_{MCM} is an MCM of N_0 and N_1, then N_{MCM} typically contains *more* tasks than N_0 and N_1. Moreover, although it is straightforward to show that any MCM is a subclass under life-cycle inheritance of any MCD ($\leq_{lc}$ is transitive; see Property 3.7), an MCM is typically larger than an MCD in terms of their numbers of tasks. Consider for example the process definitions shown in Figure 6.3. According to Definition 6.4, N_{GCD} is an MCD of the three variants shown in Figure 6.2 and N_{LCM} is an MCM of these three variants. Although $N_{LCM} \leq_{lc} N_{GCD}$, N_{LCM} has two tasks more than N_{GCD}.

Definition 6.4 raises two interesting questions:

1. Has any set of workflow process definitions always at least one MCD and at least one MCM?

2. Has any set of workflow process definitions always a GCD and an LCM?

In the remainder, we answer these questions. We show that the answer to the first question is affirmative. Unfortunately, the answer to the second question is negative. Note that, so far, we have used the terms "a GCD" and "an LCM" rather than "the GCD" and "the LCM." However, it follows immediately from Definition 6.4 that any two GCDs of a set of workflow process definitions are equivalent in the sense defined in Definition 2.25; that is, a GCD of a set of workflow process definitions is unique up to branching bisimilarity. The same is true for an LCM of a set of workflow process definitions. Therefore, in the remainder, we use the terms "the GCD" and "the LCM."

The following property is needed to prove that for any set of workflow process definitions there is at least one MCD and at least one MCM. A set of *totally* ordered (according to the life-cycle-inheritance relation $\leq_{lc}$) workflow process definitions is called a *chain*.

Property 6.5. *Let N_0 and N_1 be two workflow process definitions in $\mathcal{W}$ such that $N_0 \leq_{lc} N_1$. There is no* infinite *chain $N^0 \leq_{lc} N^1 \leq_{lc} \ldots$ of* different *(with respect to $\cong$) workflow process definitions $N^0, N^1, \ldots \in \mathcal{W}$ such that $N_0 \leq_{lc} N^0 \leq_{lc} N^1 \leq_{lc} \ldots \leq_{lc} N_1$.*

Proof. Let N and N' be two workflow process definitions in $\mathcal{W}$ such that $N \leq_{lc} N'$. The following three observations are important. First, $\alpha(N') \subseteq \alpha(N)$. Second, if $N \ncong N'$, then $\alpha(N') \subset \alpha(N)$. Third, $\alpha(N)\backslash\alpha(N')$ is finite.

Let $N^0 \leq_{lc} N^1 \leq_{lc} \ldots$ be an infinite chain of different workflow process definitions $N^0, N^1, \ldots \in \mathcal{W}$ such that $N_0 \leq_{lc} N^0 \leq_{lc} N^1 \leq_{lc} \ldots \leq_{lc} N_1$. It follows from the first two of the above observations that $\alpha(N_1) \subseteq \ldots \subset \alpha(N^1) \subset \alpha(N^0) \subseteq \alpha(N_0)$. The third observation above states that $\alpha(N_0)\backslash\alpha(N_1)$ is finite. However, this yields a contradiction, which proves the property. □

The following theorem answers the first question raised above affirmatively.

Theorem 6.6. (Existence of an MCD and an MCM) *Let $N_0, N_1, \ldots, N_{n-1}$, where n is some natural number, be n workflow process definitions in $\mathcal{W}$. There exists an MCD of $N_0, N_1, \ldots, N_{n-1}$ and there exists an MCM of $N_0, N_1, \ldots, N_{n-1}$.*

Proof. Recall Property 6.5. It states that there are no infinite chains in-between any two workflow process definitions related by $\leq_{lc}$. Consequently, to prove the existence of an MCD, it suffices to show that there exists a workflow process definition that is a superclass of all variants $N_0, N_1, \ldots, N_{n-1}$. Similarly, to prove the existence of an MCM, it suffices to show that there is a workflow process definition that is a subclass of all variants.

Let N_τ be the workflow process definition containing one task labeled τ, i.e., $N_\tau = (\{i, o\}, \{\tau\}, \{(i, \tau), (\tau, o)\}, \{(\tau, \tau)\})$. Clearly, N_τ is a superclass of any workflow process definition. Thus, N_τ is a superclass of each of the variants, which proves the existence of an MCD.

Let N_∂ be the net which is constructed from all the variants $N_0, N_1, \ldots, N_{n-1}$ as follows. The source place i of N_∂ has n output transitions, one for each variant. Each of these new transitions has a unique task label that does not occur in the alphabets of any of the variants. The source place of each variant is given a new identifier and connected as an output place to one of the n new transitions. In this way, the new transitions act as guards for the n original variants. The sink places of the n variants are simply fused together, yielding the sink place o of N_∂. Clearly, N_∂ is a subclass of each variant; by blocking all new transitions except one which is hidden, one obtains a workflow process definition branching bisimilar to one of the variants. Therefore, we conclude that also an MCM of the n variants exists. □

The answer to the first question phrased above is positive: Any set of workflow process definitions has an MCD and an MCM. Unfortunately, as already mentioned, the answer to the second question is negative. A set of workflow process definitions may have two or more different MCDs, which means that it has no GCD. Similarly, a set of workflow process definitions may have two or more different MCMs and, thus, no LCM. Consider for example the two process definitions shown in Figure 1.1. There are at least two MCDs. Both the sequential process definition consisting of task *prepare_shipment*, task *send_goods*, and task *record_shipment* and the sequential process definition consisting of task *prepare_shipment*, task *send_bill*, and task *record_shipment* are MCDs of N_0 and N_1. It is easy to verify that both workflow process definitions are MCDs. Each of them is a superclass of both N_0 and N_1 and, in both cases, there is not a smaller (with respect to $\leq_{lc}$) candidate. Similarly, the two workflow process definitions N_0 and N_1 shown in Figure 1.1 have more than one MCM. Consider the process definition N_∂ mentioned in the proof of Theorem 6.6. That is, consider a workflow process definition consisting of N_0 and N_1 and starting with two additional guard transitions. Each of the guard transitions has a unique label, say l_0 and l_1, respectively. If l_0 is blocked and l_1 hidden, then N_∂ is branching bisimilar to N_1; if l_1 is blocked and l_0 hidden, then N_∂ is branching bisimilar to N_0. Therefore, N_∂ is a subclass of both variants. There is no workflow process definition which is a subclass of both variants, a superclass of N_∂, and not branching bisimilar to N_∂. Therefore, N_∂ is an MCM of N_0 and N_1. However, the labels l_0 and l_1 were chosen arbitrarily, i.e., any pair of labels not used in N_0 and N_1 will do. Therefore, N_0 and N_1 have as many MCMs as there are combinations of labels not used in N_0 and N_1.

Based on the two variants shown in Figure 1.1, we conclude that a given set of workflow process definitions can have several MCDs and MCMs. In the example of Figure 1.1, the reason is that N_0 and N_1 do agree on the presence of the tasks *send_goods* and *send_bill*, whereas they do not agree on their ordering. However, in many situations, there is one unique (modulo branching bisimilarity) MCD, which is therefore the GCD, and one unique MCM, the LCM. For example, the three variants shown in Figure 6.2 have a GCD and an LCM, namely the nets N_{GCD} and N_{LCM} of Figure 6.3, respectively. The following theorem states necessary and sufficient requirements for the existence of a GCD and/or an LCM.

Theorem 6.7. *Let $N_0, N_1, \ldots, N_{n-1}$, where n is some natural number, be n workflow process definitions in $\mathcal{W}$.*
Workflow process definition N in $\mathcal{W}$ is the GCD of $N_0, N_1, \ldots, N_{n-1}$ if and only if

1. *$(\forall\, k : 0 \leq k < n : N_k \leq_{lc} N)$ and,*
2. *for any workflow process definition N' in $\mathcal{W}$, $(\forall\, k : 0 \leq k < n : N_k \leq_{lc} N')$ implies $N \leq_{lc} N'$.*

Workflow process definition N in $\mathcal{W}$ is the LCM of $N_0, N_1, \ldots, N_{n-1}$ if and only if

1. *$(\forall\, k : 0 \leq k < n : N \leq_{lc} N_k)$ and,*
2. *for any workflow process definition N' in $\mathcal{W}$, $(\forall\, k : 0 \leq k < n : N' \leq_{lc} N_k)$ implies $N' \leq_{lc} N$.*

Proof. The proofs of the two parts of the theorem are very similar. Therefore, we only prove the first part.

First, assume that N is the GCD of $N_0, N_1, \ldots, N_{n-1}$. It follows from Definition 6.4 (MCD,GCD) that $(\forall\, k : 0 \leq k < n : N_k \leq_{lc} N)$. Thus, N satisfies the first requirement of Theorem 6.7. To prove that it also satisfies the second requirement, assume that there exists a workflow process definition N' in $\mathcal{W}$ such that $(\forall\, k : 0 \leq k < n : N_k \leq_{lc} N')$ and $N \not\leq_{lc} N'$. It follows from Property 6.5 that N' can be chosen in such a way that it is an MCM of $N_0, N_1, \ldots, N_{n-1}$. However, by Definition 6.4-2 (GCD), this means that $N \cong N'$, which contradicts the fact that $N \not\leq_{lc} N'$ ($\leq_{lc}$ is a partial order; see Property 3.7). Thus, N satisfies also the second requirement of Theorem 6.7.

Second, let N be a workflow process definition satisfying the first pair of requirements of Theorem 6.7. Consider Definition 6.4-1 (MCD). Assume that N' is a workflow process definition such that $(\forall\, k : 0 \leq k < n : N_k \leq_{lc} N')$ and $N' \leq_{lc} N$. It follows that $N \leq_{lc} N'$, which in combination with $N' \leq_{lc} N$ implies that $N' \cong N$ ($\leq_{lc}$ is a partial order; see Property 3.7). Thus, net N satisfies the requirements in Definition 6.4-1, which means that it is an MCD. Assume that N_{MCD} is an arbitrary MCD of the n variants. It follows from the assumptions on N that $N \leq_{lc} N_{MCD}$. Consequently, Definition 6.4-1 (MCD) yields that $N \cong N_{MCD}$, which means that N satisfies Definition 6.4-2 (GCD). □

So far, we have formalized the notions of MCD, GCD, MCM, and LCM. It has been shown that MCDs and MCMs always exist, but that they are not necessarily unique. If a set of workflow process definitions has a unique MCD (MCM), then this MCD (MCM) is the GCD (LCM). The reason for studying these notions is that they are suitable to aggregate management information. That is, any MCD or MCM of a set of workflow process definitions is a suitable MI-net, as defined in Definition 6.1, for these process definitions. However, in general, it is not straightforward to determine an MCD or an MCM of a set of workflow process definitions or, when they exist, the GCD or the LCM of this set. In addition, even given an MCD, an MCM, the GCD, or the LCM, it is not always possible to find meaningful transfer rules for mapping cases in the various workflow process definitions onto such a net. However, there are situations where it is quite easy to pinpoint the GCD and/or the LCM of a set of workflow process definitions. The remainder of this subsection is devoted to explaining a number of these situations. In the next subsection, we return to the topic of finding appropriate transfer rules.

Consider some number of workflow process definitions that are variants of a single workflow process. First, if all the variants are equivalent (according to the behavioral equivalence relation $\cong$), an arbitrary variant is the GCD as well as the LCM. Second, if the variants form a *chain*, i.e., the variants are totally ordered according to the $\leq_{lc}$ relation, then the least element is the LCM and the greatest element is the GCD. Third, if one variant is a superclass of all the other variants, then this variant is the GCD. Note that the three workflow process definitions of Figure 6.2 satisfy this requirement. Process definition N_0 is a superclass of both N_1 and N_2, which means that it is the GCD of the three variants. This result conforms to our earlier claims. Fourth, if one variant is a subclass of all the other variants, then this variant is the LCM. Fifth, if two variants have no tasks in common, then the GCD equals the empty workflow process definition N_τ as introduced in the proof of Theorem 6.6. Finally, if the variants have nothing in common (i.e., with respect to internal places, transitions, and labels) and always start with a real task (i.e., a non-τ-labeled transition), then the LCM is simply the union of all workflow process definitions. The following property formalizes the above claims.

Property 6.8. *Let $N_0, N_1, \ldots, N_{n-1}$, where n is some natural number, be n workflow process definitions in $\mathcal{W}$.*

1. *If $N_0 \cong N_1 \cong \ldots \cong N_{n-1}$, then, for any k with $0 \leq k < n$, N_k is both the GCD and the LCM of $N_0, N_1, \ldots, N_{n-1}$.*
2. *If $N_0 \leq_{lc} N_1 \leq_{lc} \ldots \leq_{lc} N_{n-1}$, then N_0 is the LCM and N_{n-1} is the GCD of $N_0, N_1, \ldots, N_{n-1}$.*
3. *If, for all k with $0 \leq k < n$, $N_k \leq_{lc} N_0$, then N_0 is the GCD of $N_0, N_1, \ldots, N_{n-1}$.*
4. *If, for all k with $0 \leq k < n$, $N_0 \leq_{lc} N_k$, then N_0 is the LCM of $N_0, N_1, \ldots, N_{n-1}$.*
5. *If, for some j and k with $0 \leq j < k < n$, $\alpha(N_j) \cap \alpha(N_k) = \emptyset$, then $N_\tau = (\{i, o\}, \{\tau\}, \{(i, \tau), (\tau, o)\}, \{(\tau, \tau)\})$ is the GCD of $N_0, N_1, \ldots, N_{n-1}$.*
6. *If, for all j and k with $0 \leq j < k < n$, $\alpha(N_j) \cap \alpha(N_k) = \emptyset$ and $(P_j \cup T_j) \cap (P_k \cup T_k) = \{i, o\}$ and, for all k with $0 \leq k < n$ and all transitions $t \in i \overset{N_k}{\bullet}$, t has a label different from τ, then $N_\partial = \bigcup_{0 \leq k < n} N_k$ is the LCM of $N_0, N_1, \ldots, N_{n-1}$.*

Proof. The first four properties follow immediately from Theorem 6.7.

To prove the fifth property, first, observe that N_τ is a superclass under life-cycle inheritance of all n variants. (See Definition 3.4-4 (Life-cycle inheritance); clearly, hiding all tasks in a variant yields a process equivalent to N_τ.) Second, let N' be an arbitrary superclass of $N_0, N_1, \ldots, N_{n-1}$. Consider two variants N_j and N_k, with $0 \leq j < k < n$, such that $\alpha(N_j) \cap \alpha(N_k) = \emptyset$. Since $N_j \leq_{lc} N'$, it follows that $\alpha(N') \subseteq \alpha(N_j)$; similarly, $\alpha(N') \subseteq \alpha(N_k)$. Hence, it follows that $\alpha(N') \subseteq \alpha(N_j) \cap \alpha(N_k) = \emptyset$, which means that $\alpha(N') = \emptyset$. Consequently, $N' \cong N_\tau$, which means that $N_\tau \leq_{lc} N'$. Hence, by Theorem 6.7, we conclude that N_τ is the GCD of the set of variants N_0 through N_{n-1}.

To prove the last property, we first show that N_∂ is a subclass of each of the variants. Consider a variant N_k, for some k with $0 \leq k < n$. Since for all j with $0 \leq j < n$ and $j \neq k$, $\alpha(N_j) \cap \alpha(N_k) = \emptyset$, $(P_j \cup T_j) \cap (P_k \cup T_k) = \{i, o\}$, and all transitions $t \in i \overset{N_j}{\bullet}$ have a label different from τ, blocking all transitions in $i \overset{N_\partial}{\bullet} \backslash i \overset{N_k}{\bullet}$ in N_∂, yields a process branching bisimilar to N_k. Hence, $N_\partial \leq_{lc} N_k$, which means that it is a subclass of all n variants. Second, we prove that any workflow process definition N' in $\mathcal{W}$ that is a subclass of all the variants is also a subclass of N_∂. Assume that $N' \in \mathcal{W}$ is a subclass of all variants. Let, for all k with $0 \leq k < n$, I_k and H_k be sets of task labels such that $(\tau_{I_k} \circ \partial_{H_k}(N'), [i]) \sim_b (N_k, [i])$ (see Definition 3.4-4 (Life-cycle inheritance)). Let $I = \bigcup_{0 \leq k < n} I_k$ and $H = \bigcup_{0 \leq k < n} H_k$. Clearly, $(\tau_I \circ \partial_H(N'), [i]) \sim_b (N_\partial, [i])$, because each label in H or I appears in the alphabet of precisely one of the n variants. Hence, $N' \leq_{lc} N_\partial$. Combining the two results derived so far, it follows from Theorem 6.7 that N_∂ is the LCM of the set of variants N_0 through N_{n-1}. $\square$

6.3 Inheritance-preserving transformation rules and management information

In Section 6.1, the notion of an MI-net has been introduced as a means to collect aggregate management information on the status of cases in a number of variants of a workflow process. It is essential that a set of total and valid transfer rules is available to map information of running cases onto an MI-net. Section 6.2 has introduced the notions of an MCD/GCD and an MCM/LCM of a number of workflow process definitions. It has been argued that an MCD/GCD and an MCM/LCM are MI-nets that are good candidates for collecting aggregate management information. However, it is not always easy to determine an MCD, an MCM, the GCD, or the LCM. As indicated in Section 6.2, there may even be situations where the GCD or LCM does not exist.

The mapping of running cases in different variants of the workflow process onto a suitable MI-net is crucial. Unfortunately, it is not always straightforward to obtain a useful set of transfer rules. However, in Section 5, it has been shown that it is always possible to transfer a case from one workflow process definition to another one if the latter is constructed from the former by means of one of the inheritance-preserving transformation rules of Section 3.2. Thus, the inheritance-preserving transformation rules and accompanying transfer rules can also be used to extract aggregate management information.

Observation 6.9. *Consider a set of workflow process definitions that are created from each other by means of the inheritance-preserving transformation rules presented in Section 3.2 (in both directions). For each pair of elements of this set, the transfer rules of Section 5 can be used to construct a total valid transfer rule which maps cases from one element of this pair to the other element.*

Observation 6.9 implies that *any* workflow process definition of a set of process definitions that are created from each other by means of the inheritance-preserving transformation rules of Section 3.2 forms a meaningful MI-net. Consider again the three workflow process definitions in Figure 6.2. Workflow process definition N_1 can be obtained from N_0 by means of inheritance-preserving transformation rule *PJ3S* of Theorem 3.19, whereas N_2 can be obtained from N_0 with transformation rule *PJS* of Theorem 3.16. This means that all cases in N_1 and N_2 can be mapped onto N_0 by means of transfer rules r_{PJ3S}^{-1} Theorem 5.24 and r_{PJS}^{-1} of Theorem 5.22, respectively. However, it is more interesting to see how cases can be mapped onto N_1 and N_2. Assume that N_1 is used as an MI-net. All cases in N_0 can be mapped onto N_1 by means of the rule $r_{PJ3S,P}$ of Theorem 5.16. (Recall that we have explicitly excluded rule $r_{PJ3S,C}$ in the context of aggregating management information.) Cases in N_2 can be mapped onto N_1 by means of the composite transfer rule $r_{PJ3S,P} \circ r_{PJS}^{-1}$. That is, net N_0 is used as an intermediate to map cases from N_2 onto N_1. Similarly, all cases can be mapped onto N_2 by means of transfer rules r_{PJS} of Theorem 5.11 and $r_{PJS} \circ r_{PJ3S}^{-1}$.

Observation 6.9 has several important consequences. Consider a set of workflow process definitions satisfying the requirement in Observation 6.9. As already mentioned, any process definition in this set can be chosen as an MI-net. The transfer rules of Section 5 based on the four inheritance-preserving transformation rules (i.e., r_{PTS}, r_{PPS}, r_{PJS}, $r_{PJ3S,P}$, $r_{PTS,C}^{-1}$, $r_{PTS,P}^{-1}$, r_{PPS}^{-1}, r_{PJS}^{-1}, and r_{PJ3S}^{-1}) provide mappings for running cases in any of the workflow process definitions onto states in the MI-net. These states are as close to the actual states of the cases as possible, which is very important for the quality of the management information. Note that the chosen MI-net is not necessarily an MCD or an MCM of the set of process definitions (see also the example discussed above and illustrated in Figures 6.2 and 6.3). However, if the set of workflow process definitions satisfies any of the conditions of Property 6.8, then a suitable choice for the MI-net yields the GCD or the LCM (see, again, the example illustrated in Figures 6.2 and 6.3).

Figure 6.10 illustrates a slightly larger example. It shows four workflow process definitions. The two process definitions N_0 and N_1 in the middle are two variants of the complaints-handling process.

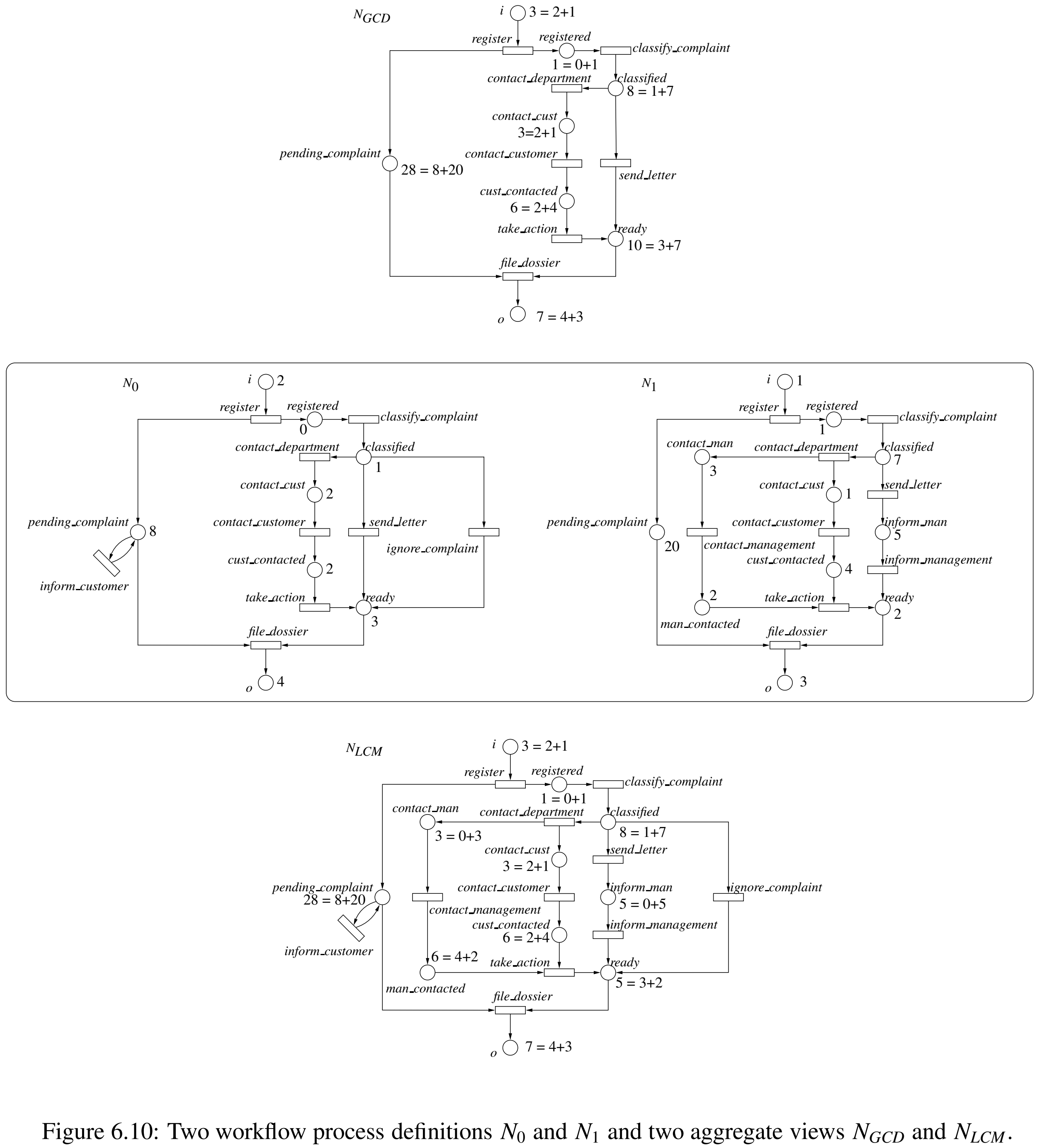

Figure 6.10: Two workflow process definitions N_0 and N_1 and two aggregate views N_{GCD} and N_{LCM}.

These two variants hold 38 cases, 14 in N_0 and 24 in N_1. The other two process definitions are MI-nets. Net N_{GCD} (top) is the GCD of N_0 and N_1; net N_{LCM} (bottom) is the LCM of N_0 and N_1. Net N_{GCD} can be obtained from N_0 by means of transformation rules *PPS* and *PTS*, both applied in reverse direction; N_{GCD} is obtained from N_1 by means of rules *PJS* and *PJ3S* in reverse direction. Furthermore, net N_0 yields N_{LCM} via rules *PJS* and *PJ3S*, whereas N_1 yields N_{LCM} via rules *PPS* and *PTS*.

The transfer rules of Section 5 are used to map cases of the variants N_0 and N_1 onto the two MI-nets; the composition of r^{-1}_{PPS} and $r^{-1}_{PTS,C}$ (or $r^{-1}_{PTS,P}$ wich equals $r^{-1}_{PTS,C}$ for this example) is used to map cases of N_0 onto N_{GCD}, the composition of r^{-1}_{PJS} and r^{-1}_{PJ3S} is used to map cases of N_1 onto N_{GCD}, the composition of r_{PJS} and $r_{PJ3S,P}$ is used to map cases of N_0 onto N_{LCM}, and the composition of r_{PPS} and r_{PTS} is used to map cases of N_1 onto N_{LCM}.

Consider, for example, place *ready* in N_{GCD}. The label $10 = 3+7$ indicates that ten of the 38 cases are in the state corresponding to place *ready*. In N_0, three cases are ready and, in N_1, two cases are ready, i.e., just five cases are actually ready. However, there are five cases in the state corresponding to *inform_man* in N_1. If we abstract from task *inform_management*, these cases are also ready. This brings the total to ten cases in state ready in N_{GCD}. Note that, in N_{LCM}, there are just five cases in a state corresponding to place *ready*, because the LCM distinguishes between *inform_man* and *ready*. Figure 6.10 is a good example illustrating that the LCM is more complex and contains more detailed information, whereas the GCD is more succinct and only contains information which is relevant for all variants. Which one is most suitable as an MI-net depends on the context.

6.4 Management information in the workflow-management domain

To end this section on management information, let us return to Section 4. In that section, we have discussed the relevance of inheritance in four domains: *ad-hoc change*, *evolutionary change*, *workflow templates*, and *E-commerce*. In the remainder, we discuss for each of these domains possible approaches for obtaining aggregate management information using the inheritance-preserving transformation rules and the transfer rules presented in this paper.

Ad-hoc change typically results in many slightly different variants of a predefined workflow process. These variants are usually the result of an error, a rare event, or special demands of a customer. The predefined workflow process can be seen as a template. If all variants are constructed by extending the template workflow using the inheritance-preserving transformation rules and the template itself is also a variant, then the template workflow is the GCD of all variants (see Property 6.8-3). If the variants are constructed by applying the inheritance-preserving transformation rules in the reverse direction only and the template itself is also a variant, then the template workflow is the LCM of all variants (see Property 6.8-4). If the variants are constructed by applying the inheritance-preserving transformation rules in both directions, the template workflow is not the GCD nor the LCM but it is still a suitable MI-net for presenting aggregate management information. If change is restricted according to the inheritance-preserving transformation rules, then the transfer rules of Section 5 can be used to obtain transfer rules from the ad-hoc variants to the template workflow process (see Observation 6.9).

Evolutionary change typically results in a limited set of versions of a workflow process. If every time a change occurs, all cases are transferred immediately, there is just one active version. Only if transfers are postponed (e.g., transfer rule $r_\emptyset$), there is a need to aggregate management information. In case of evolutionary change, the most recent version of a workflow process is the most likely candidate for presenting aggregate management information. If all changes are restricted to the inheritance-preserving transformation rules, it is no problem to map the cases onto the most recent version of the workflow process (Observation 6.9). Note that, if all changes in the past were *extensions* (i.e., the transformation rules were only applied in the forward direction), then the most recent version of the workflow process definition is the LCM of all variants (see Property 6.8-2).

When using a workflow template as the starting point for designing workflows, the template is the most likely candidate for projecting aggregate management information. Again, by restricting modifications of the template to the four inheritance-preserving transformation rules presented in this paper, all cases can be mapped onto the workflow template without any problems.

For E-commerce, it is important that business partners agree on some common workflow process (see Section 4.4). For each of the business partners, it is useful to have aggregated management information at the level of the common workflow process. In Section 4.4, it has been suggested that each of the local workflow processes should be a subclass of (part of) the common workflow process under projection inheritance. If local extensions are restricted to transformation rules *PPS*, *PJS*, and *PJ3S*, then the transfer rules r_{PPS}^{-1}, r_{PJS}^{-1}, and r_{PJ3S}^{-1} can be used to map cases onto the common process definition. Moreover, the common process definition is, under certain restrictions, the GCD of the workflow processes perceived by the business partners (i.e., local and global view).

7 Tool support

In the preceding sections, we have shown that inheritance concepts can be used to tackle many of the problems related to ad-hoc change and evolutionary change of workflow processes. Moreover, the concepts can be used to enhance the application of workflow templates and may be beneficial in the design and enactment of interorganizational workflows (see Section 4.4). Unfortunately, today's workflow management systems do not support workflow inheritance as discussed in this paper. Some workflow management systems have adopted object-oriented concepts. For example, InConcert [39] allows for building workflow class hierarchies. However, in these class hierarchies, inheritance is restricted to the static interface (i.e., attributes and/or is-part-of relationships). To our knowledge, there is not a single workflow management system taking the dynamics of the workflow process into account when defining inheritance. In this section, we briefly discuss how the results presented in this paper can be used to aid existing workflow management systems.

In the remainder of this section, first, we describe Woflan which allows for the verification of soundness. (Recall that soundness is pivotal to the notions of inheritance, the transformation rules, and the transfer rules.) Then, we discuss tool support for the inheritance notions introduced in Section 3. Woflan can be used to check relationships under any of the four inheritance relations introduced in this section. Finally, we consider ways to integrate change facilities, i.e., services to support dynamic change and to construct aggregate management information, in existing workflow management systems.

7.1 Verifying soundness

Throughout this paper, we considered workflow process definitions. Recall that a workflow process definition is a sound WF-net. That is, a workflow process definition determines not an arbitrary process but a process with desirable properties such as proper completion, absence of deadlock, etc. (see Definitions 2.19 (WF-net) and 2.22 (Soundness)). Most of today's workflow management systems can only enact workflow processes having these properties. However, they do not support advanced techniques to verify the correctness of workflow process definitions [2]. At design-time, there are hardly any checks to verify whether the properties stated in Definitions 2.19 and 2.22 are fulfilled.

Violations of these properties typically result in serious run-time errors such as deadlocks or livelocks. Contemporary workflow management systems typically restrict themselves to a number of (trivial) syntactical checks. Therefore, serious errors such as deadlocks and livelocks may remain undetected. This means that an erroneous workflow may go into production, thus causing dramatic problems for the organization. An erroneous workflow may lead to extra work, legal problems, angry customers, managerial problems, and depressed employees. Therefore, it is important to verify the correctness of a workflow process definition *before* it becomes operational. If there are frequent ad-hoc or evolutionary changes, then the role of verification becomes even more important. This is the reason that we developed *Woflan* (WOrkFLow ANalyzer) [58, 59]. Woflan is a stand-alone verification tool specifically designed for workflow analysis. Woflan is *product independent*, i.e., it is possible to analyze processes designed with various workflow products of different vendors. Woflan is able to handle *complex workflows* with up to hundreds of tasks. Woflan provides to-the-point *diagnostic information* for repairing the errors detected. Pivotal to Woflan is the notion of soundness as defined in Definition 2.22. In fact, Woflan uses a slightly weaker version of soundness where the safeness requirement is omitted (cf. [2, 59]). However, Woflan also analyzes the stronger notion used in this paper. The soundness notion expresses the minimal requirements any workflow should satisfy and includes properties such as proper termination and the absence of deadlock and livelocks. The current version of Woflan can analyze workflows designed with the following four workflow products: COSA, Staffware, METEOR, and Protos. COSA [56] is one of the leading workflow management systems on the Dutch workflow market. COSA uses Petri nets as a modeling language and thus allows for the modeling and enactment of complex workflow processes which use advanced routing constructs. However, COSA does not support verification. Fortunately, Woflan can analyze any workflow process definition constructed by using CONE (COSA Network Editor), the design tool of the COSA system. Woflan can also import process definitions made with Staffware [7, 57], METEOR [55], or Protos [48]. Staffware is one of the most widespread workflow management systems in the world. METEOR is a workflow management system based on CORBA and supports transactional workflows ([31]). Protos is a Business-Process-Reengineering tool which can be used to (re)design and document workflow processes.

To illustrate the use of Woflan, consider the WF-net shown in Figure 2.21. Figure 7.1 shows this workflow modeled with COSA and Figure 7.2 shows some of the diagnostics provided by Woflan when analyzing this workflow. Woflan reports that the workflow modeled with COSA is not sound and that the connection between tasks *ignore_complaint* and *inform_management* is the source of the error. (Note that the boundedness and safeness property in the diagnosis of Woflan refer to the short-circuited workflow net; see Definition 2.19 (WF-net) and Theorem 2.23 (Characterization of soundness).) For more information on Woflan, we refer to [59]; the interested reader can also download a version of Woflan via the World-Wide-Web [58].

Note that the four inheritance-preserving transformation rules presented in Section 3.2 appear to reduce the need for a tool like Woflan: The four rules preserve soundness. However, the inheritance-preserving transformation rules require essential parts of the functionality implemented in Woflan. Consider for example requirement 4 in Theorem 3.12, requirements 4 and 6 in Theorem 3.14, requirement 3 in Theorem 3.16, and requirements 4, 6, and 7 in Theorem 3.19. These requirements need to be checked via algorithms like those implemented in Woflan. As explained in the next subsection, Woflan provides an excellent basis to incorporate support for inheritance.

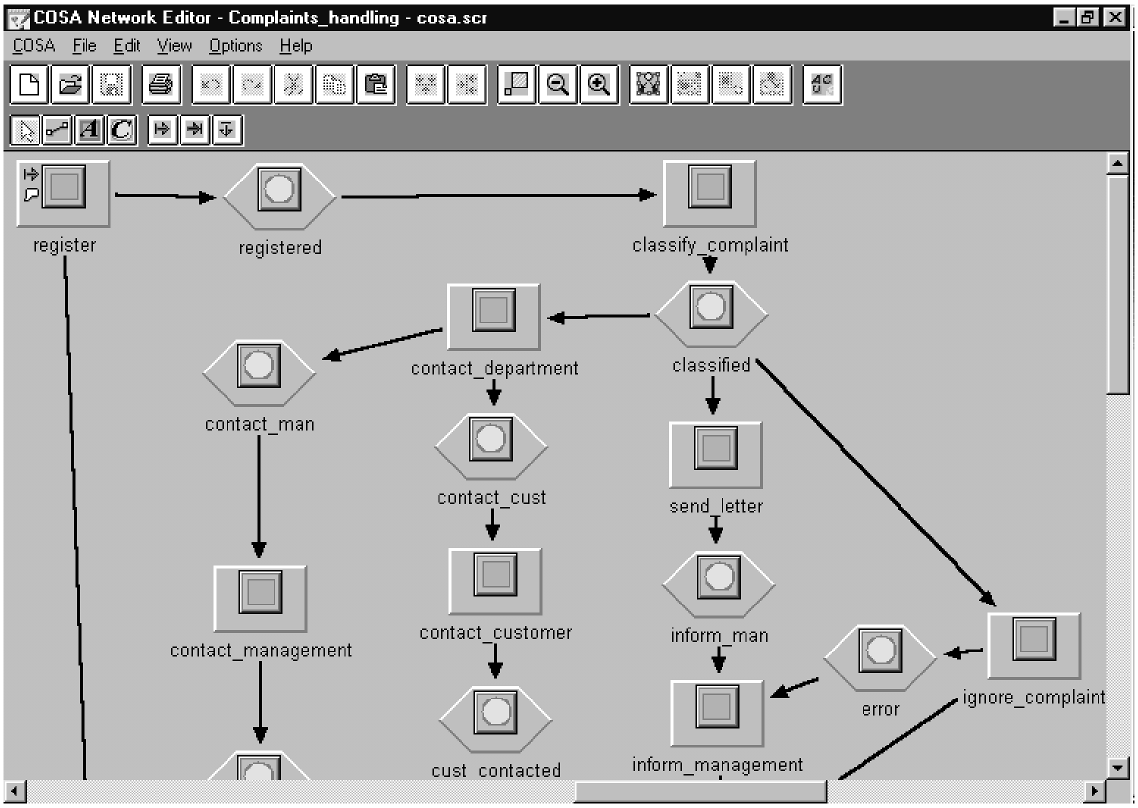

Figure 7.1: The COSA design of the erroneous workflow process definition shown in Figure 2.21.

7.2 Supporting inheritance

In Section 4, we have shown several application areas where it is desirable to limit possible changes by imposing inheritance relationships, e.g., the designed workflow process definition should be extended in such a way that the result is a subclass of a predefined workflow process definition (e.g., a workflow template or existing workflow) under life-cycle inheritance. Recall that we have defined four inheritance relations: protocol/projection inheritance, protocol inheritance, projection inheritance, and life-cycle inheritance. Basically, there are two ways to support these inheritance relations.

1. *Enumerative verification method*
For any two workflow process definitions, it is decidable whether one workflow process definition is a subclass of the other workflow process definition under one of the four inheritance relations of Definition 3.4. By comparing the state spaces of two process definitions, it is possible to decide whether the process definitions are branching bisimilar. Therefore, a brute-force approach can be used by systematically blocking and hiding tasks, enumerating all reachable states of the resulting nets, and comparing the state spaces. There are several tools that can check branching bisimilarity using enumerative methods. It is well-known that deciding whether two finite processes are branching bisimilar can be done in polynomial time, where the size of the problem is defined as the number of states and transitions of the two processes [35]. However, even a workflow process with a limited number of tasks can have many states. Therefore, there are two practical problems when using a separate verification tool based on enumerative methods. First, it is difficult to provide an interface between the workflow editor

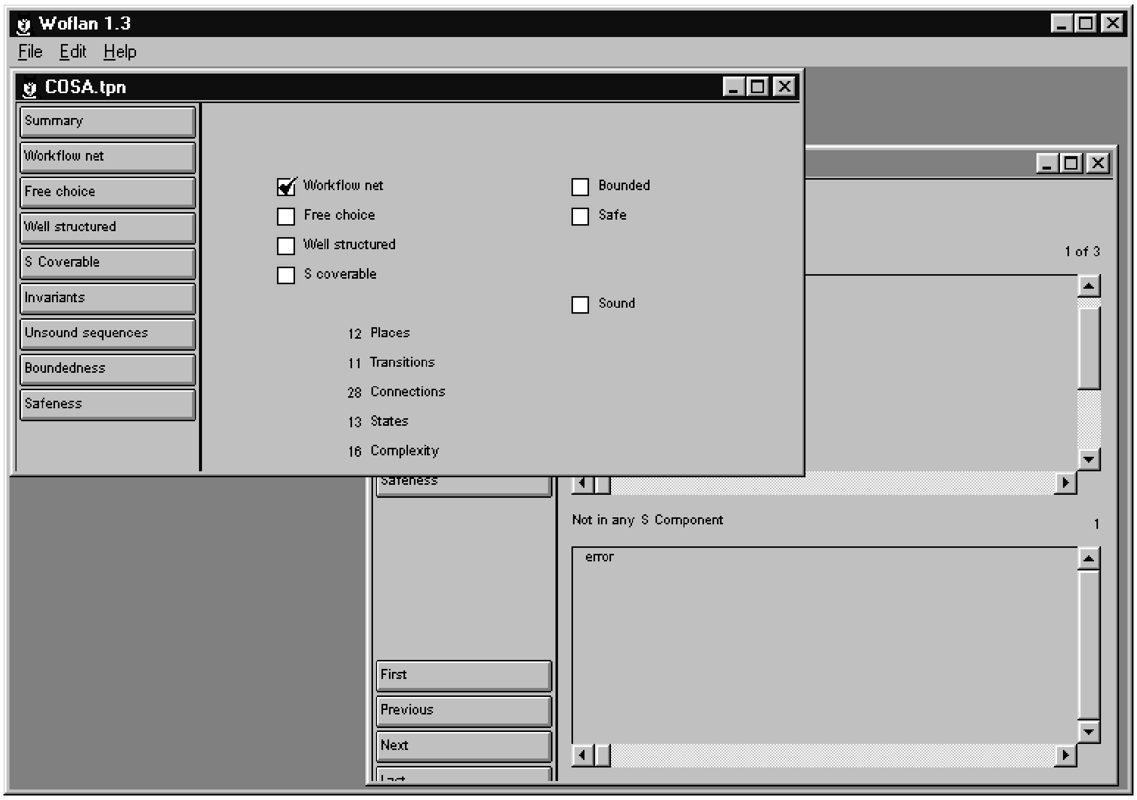

Figure 7.2: Some of the diagnostics provided by Woflan when verifying the correctness of the COSA design shown in Figure 7.1.

(i.e., the workflow design tool) and the verification tool. The workflow editor has to construct the state space typically containing thousands of states and send it to the verification tool. Second, it is very difficult to translate the results generated by the verification tool to diagnostics understandable by the workflow designer.

2. *Workflow editor supporting inheritance-preserving transformation rules*
   In Section 3.2, we have identified four inheritance-preserving transformation rules. Instead of using an enumerative method to verify inheritance relations afterwards, it is possible to limit the changes in the workflow editor to the four inheritance-preserving transformation rules identified in this paper. Note that the transformation rules correspond to the design constructs typically used when constructing/adapting a workflow process definition. Using an editor augmented with these rules has two benefits. First, using the rules instead of an enumerative method is more efficient from a computational point-of-view. Second, the user is forced to make correct designs with respect to the selected inheritance relation (correctness by design). Therefore, there is no need to provide diagnostics to locate the source of an error. Unfortunately, the editors of current workflow management systems do not provide facilities to enforce design rules and the conditions for the transformation rules are quite complex to check by tools not dedicated to Petri-net analysis. Therefore, it will not be easy to extend the existing workflow tools with inheritance-preserving transformation rules.

The ideal situation would be an editor which automatically checks inheritance relationships or limits change to the inheritance-preserving transformation rules. At the moment, such tools are missing. Unfortunately, it is also not likely that workflow management systems will provide sophisticated editors supporting the inheritance notions in the very near future. Therefore, it is useful to extend Woflan with support for the inheritance notions of this paper. Recall that Woflan can import process definitions from several workflow tools. Thus, Woflan can provide tool-independent support for inheritance. In principle, it is possible to implement both enumerative verification of inheritance relationships and support for the inheritance-preserving transformation rules in Woflan.

The current version of Woflan implements the enumerative approach based on the algorithm of [35]. It can check whether one workflow process definition is a subclass of another workflow process definition under any of the four inheritance relations of Definition 3.4. For protocol inheritance, projection inheritance, and protocol/projection inheritance, this check is quite efficient, i.e., polynomial in the number of states and transitions of the two workflow processes. In the current version of Woflan, it is more involved to check life-cycle inheritance. At the moment, Woflan only supports a brute-force approach which (in the worst case) checks all possible partitions of new tasks (i.e., tasks present in one workflow process definition but not in the other one) into sets of tasks that need to be blocked and those that need to be hidden (see Definition 3.4-4). A workflow designer can use the current version of Woflan to verify whether or not a proposed change of a workflow process is captured by any of the four inheritance relations. Note that this approach only partly solves the problems related to enumerative verification mentioned above. Woflan provides tool-independent support, but state spaces may still become very large and it might be difficult to provide useful diagnostics in case a desired subclass relationship does not exist.

The algorithms on which Woflan is based can also be used to verify most of the requirements for the inheritance-preserving transformation rules. Thus, it is possible to extend Woflan with support for the transformation rules in a relatively straightforward way. However, in order to be useful with existing workflow tools, a workflow designer must translate a transformation verified by such an extended version of Woflan to the workflow model used by the tool. Such a translation may be error-prone if the modeling language of the workflow tool is not closely related to Petri nets. As an alternative, the transformation rules can also be used as a *method* to be employed in combination with existing workflow tools. For this purpose, it is useful to translate the rules to the specific modeling language of the workflow tool. This means that the workflow designer has to check the appropriate conditions before making a change. If necessary, verification support could be provided by Woflan and/or the rules can be simplified by further restrictions. Currently, such a method appears to be the most promising way to enable workflow designers to benefit from the transformation rules presented in this paper using current technology.

7.3 Supporting dynamic change

Most of today's workflow management systems provide a versioning mechanism, i.e., it is possible to enact multiple versions of the same workflow process at the same time. However, each case (i.e., workflow instance) refers to one version and it is not possible to migrate a case from one version to another. In addition, such a mechanism is not suitable for ad-hoc change. Some workflow management systems such as InConcert [39] and Ensemble [30] provide support for ad-hoc changes, i.e., while executing a case it is possible to adapt the corresponding process definition; each case has a private copy of the workflow process definition which can be modified without any problems.

None of today's commercial workflow management systems support dynamic change, i.e., it is not possible to transfer a case from one process definition to another. Yet, for many applications such dynamic changes are a necessity. In Section 5, we presented several transfer rules under the assumption that changes are limited to the application of the inheritance-preserving transformation rules of Section 3.2 (both directions). To support the transfer of cases from one version of a process to another, the workflow enactment service ([44]) needs to extended. If change is limited to the inheritance-preserving transformation rules, the implementation of a transfer facility is rather straightforward since there is no need to postpone transfers (i.e., there is always just one active version of the workflow process). Note, however, that this assumption implies that the workflow management system includes some support for the inheritance rules, as discussed in the previous subsection. The transfer of cases can be handled by the workflow engine(s) or by a separate service. If the workflow engine is extended with a transfer facility, then the engine is notified every time there is a new version of a workflow process. For each case which is not active (i.e., no tasks are being executed), the transfer is a simple database update: Change the reference of the case and create a new workflow state (i.e., marking). If a task is being executed (for a case which needs to be transferred), the transfer is delayed until completion of the task or the running task is aborted and rolled back before the case is transferred. If a separate service is used to transfer the cases (i.e., a service not integrated in the engine), all relevant cases need to be blocked (i.e., all instances which need to be transferred are frozen) to avoid concurrency problems.

7.4 Providing aggregate management information

If there are multiple versions or variants of the same workflow process, it is desirable to have an aggregated view of the work-in-progress, i.e., condensed management information showing the statuses of all cases in one diagram (i.e., an MI-net). In Section 6, we have shown that if change is limited to the inheritance-preserving transformation rules (applied in both directions), then it is possible to construct such a view. For this purpose, the following information is needed: The states of all cases involved (including version/variant information), the transformation rules used to move from one version or variant to another, and the MI-net. For a suitably chosen MI-net, the transfer rules can be calculated automatically and all cases can be mapped onto a single diagram. Clearly, today's workflow management systems do not provide such a facility and show aggregate management information at the level of versions/variants rather than processes. (In fact, many workflow management systems provide hardly any management information.) Although the implementation of such a facility is far from trivial, there are two circumstances which simplify the realization of the ideas presented in Section 6. First, the information needed to distill the management information can be extracted without interfering with the enactment service, because cases are not actually transferred. Second, much of the functionality needed to implement dynamic change (e.g., the transfer rules) can be used for this facility.

8 Conclusion

This paper tackles two notorious problems related to adaptive workflow: (1) supporting *dynamic change* and (2) providing *management information* at the right aggregation level. The solution is based on an approach using inheritance. Since the inheritance notions used in this paper focus on the dynamic behavior of processes rather than their static structure, they are of particular relevance for workflow management. We have provided four inheritance relations (protocol/projection inheritance, protocol inheritance, projection inheritance, and life-cycle inheritance), four inheritance-preserving transformation rules which can be applied in two directions (*PTS*, *PPS*, *PJS*, and *PJ3S*), and ten transfer rules (r_{PTS}, r_{PPS}, r_{PJS}, $r_{PJ3S,C}$, $r_{PJ3S,P}$, $r^{-1}_{PTS,C}$, $r^{-1}_{PTS,P}$, r^{-1}_{PPS}, r^{-1}_{PJS}, and r^{-1}_{PJ3S}). The transformation rules can be used to restrict changes in workflow process definitions in such a way that the new workflow process definition inherits certain properties of the old workflow process definition. Such restrictions are useful when dealing with ad-hoc workflow, evolutionary change, workflow templates, and interorganizational workflows. Moreover, the transformation rules combined with the transfer rules enable dynamic change and aggregation of management information. If process changes are restricted to the transformation rules, then the typical problems related to adaptive workflow can be avoided. The transfer rules which are used to transfer cases from one workflow process definition to another can also be used to generate condensed management information showing an aggregate view of the work-in-progress. The inheritance notions are interesting both from a theoretical and a practical perspective. On the one hand, the inheritance relations lead to interesting concepts such as the GCD and the LCM of a set of process definitions. On the other hand, they provide concrete solutions for problems today's workflow management systems are faced with.

An interesting topic for future research is the application of the inheritance rules in various domains. We already mentioned the application of projection inheritance to E-commerce. Another application would be the integration of our inheritance concepts into component-based software architectures. A future challenge is also to deal with the dynamic-change problem in case there is no inheritance relationship between the old and the new process definition. One solution is to merge the approach presented in this paper with the techniques of [10, 23, 26, 27] as explained in Section 5.5. Such a combined approach identifies changes that are captured by our inheritance-preserving transformation rules as well as regions with changes that are not captured by these rules. Changes inside these regions are handled using the techniques presented in [10, 23, 26, 27]. A final challenge is to further develop tool support for the inheritance notions of this paper. As explained in Section 7.2, our tool Woflan provides a good starting point. The ultimate goal is to integrate the inheritance notions in an industrial workflow management system that supports dynamic change as well as the aggregation of management information.

Acknowledgment We want to thank Robert van der Toorn, Eric Verbeek and an anonymous referee for their careful reading of and useful comments on early versions of this paper. We thank Eric Verbeek also for his valuable contribution to the on-going development of Woflan.

References

1. W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997, Proceedings*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426, Toulouse, France, June 1997. Springer, Berlin, Germany, 1997.

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

3. W.M.P. van der Aalst. Process-oriented Architectures for Electronic Commerce and Interorganizational Workflow. *Information Systems*, 24(8):639–671, 2000.

4. W.M.P. van der Aalst and T. Basten. Life-cycle Inheritance: A Petri-net-based Approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997, Proceedings*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81, Toulouse, France, June 1997. Springer, Berlin, Germany, 1997.

5. W.M.P. van der Aalst, T. Basten, H.M.W. Verbeek, P.A.C. Verkoulen, and M. Voorhoeve. Adaptive Workflow: On the Interplay between Flexibility and Support. In J. Filipe, editor, *Enterprise Information Systems*, pages 61–68. Kluwer Academic Publishers, Norwell, 2000.

6. W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management: Models, Techniques, and Empirical Studies*. Springer, Berlin, Germany, 2000.

7. W.M.P. van der Aalst and A.H.M. ter Hofstede. Verification of Workflow Task Structures: A Petri-net-based Approach. *Information Systems*, 2000. To appear.

8. W.M.P. van der Aalst, G. De Michelis, and C.A. Ellis, editors. *Workflow Management: Net-based Concepts, Models, Techniques, and Tools (WFM'98), Proceedings*, Lisbon, Portugal, June 1998. Eindhoven University of Technology, Eindhoven, The Netherlands, Computing Science Report 98/7, 1998.

9. N.R. Adam, V. Atluri, and W.K. Huang. Modeling and Analysis of Workflows using Petri Nets. *Journal of Intelligent Information Systems*, 10(2):131–158, March 1998.

10. A. Agostini and G. De Michelis. Simple Workflow Models. In W.M.P. van der Aalst, G. De Michelis, and C.A. Ellis, editors, *Workflow Management: Net-based Concepts, Models, Techniques, and Tools (WFM'98), Proceedings*, pages 146–164, Lisbon, Portugal, June 1998. Eindhoven University of Technology, Eindhoven, The Netherlands, Computing Science Report 98/7, 1998.

11. Baan. *BaanWorkflow Product Description*. Baan Company, Barneveld, The Netherlands, 1998.

12. J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1990.

13. T. Basten. Branching Bisimilarity is an Equivalence indeed! *Information Processing Letters*, 58(3):141–147, May 1996.

14. T. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, December 1998.

15. T. Basten and W.M.P. van der Aalst. Inheritance of Behavior. Computing Science Report 99/17, Eindhoven University of Technology, Eindhoven, The Netherlands, November 1999.

16. T. Basten and W.M.P. van der Aalst. Inheritance of Dynamic Behavior: Development of a Groupware Editor. In G. Agha, F. De Cindo, and G. Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, Lecture Notes in Computer Science, Advances in Petri Nets. Springer, Berlin, Germany, 2000. To appear.

17. G. Berthelot. Transformations and Decompositions of Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets 1986, Part I: Petri Nets, Central Models and their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 360–376. Springer, Berlin, Germany, 1987.

18. E. Bertino and L. Martino. *Object-Oriented Database Systems: Concepts and Architectures*. Addison-Wesley, 1993.

19. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

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

21. S. Christensen and K. Mortensen. World of Petri Nets. http://www.daimi.au.dk/PetriNets/.

22. J.M. Colom and M. Silva. Improving the Linearly Based Characterization of P/T Nets. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 113–146. Springer, Berlin, Germany, 1990.

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

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

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

26. C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic Change within Workflow Systems. In N. Comstock, C.A. Ellis, R. Kling, J. Mylopoulos, and S. Kaplan, editors, *Conference on Organizational Computing Systems, Proceedings*, pages 10 – 21, Milpitas, California, August 1995. ACM Press, New York, 1995.

27. C.A. Ellis, K. Keddara, and J. Wainer. Modeling Workflow Dynamic Changes Using Timed Hybrid Flow Nets. In W.M.P. van der Aalst, G. De Michelis, and C.A. Ellis, editors, *Workflow Management: Net-based Concepts, Models, Techniques, and Tools (WFM'98), Proceedings*, pages 109–128, Lisbon, Portugal, June 1998. Eindhoven University of Technology, Eindhoven, The Netherlands, Computing Science Report 98/7, 1998.

28. C.A. Ellis and G.J. Nutt. Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993, Proceedings*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16, Chicago, Illinois, June 1993. Springer, Berlin, Germany, 1993.

29. J. Esparza and M. Nielsen. Decibility Issues for Petri Nets - A Survey. *Journal of Information Processing and Cybernetics*, 30(3):143–160, 1994.

30. FileNET. *Ensemble User Guide*. FileNET Corporation, Costa Mesa, California, 1998.

31. D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

32. R.J. van Glabbeek. The Linear Time – Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves (extended abstract). In E. Best, editor, *CONCUR '93, 4th. International Conference on Concurrency Theory, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81, Hildesheim, Germany, August 1993. Springer, Berlin, Germany, 1993.

33. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89: Proceedings of the IFIP 11th. World Computer Congress*, pages 613–618, San Francisco, California, August/September 1989. Elsevier Science Publishers B.V., North-Holland, 1989.

34. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

35. J.F. Groote and F.W. Vaandrager. An Efficient Algorithm for Branching and Stuttering Equivalence. In M.S. Paterson, editor, *Automata, Languages and Programming, 17th. International Colloquium, Proceedings*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638, Warwick, UK, July 1990. Springer, Berlin, Germany, 1990.

36. Y. Han and A. Sheth. On Adaptive Workflow Modeling. In *Information Systems Analysis and Synthesis, 4th. International Conference, Proceedings*, pages 108–116, Orlando, Florida, July 1998.

37. P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein, and M. Teschke. A Comprehensive Approach to Flexibility in Workflow Management Systems. In G. Georgakopoulos, W. Prinz, and A.L. Wolf, editors, *Work Activities Coordination and Collaboration (WACC'99), Proceedings*, pages 79–88, San Francisco, California, February 1999. ACM press, New York, 1999.

38. J.A. Hernandez. *The SAP R/3 Handbook*. McGraw-Hill, 1997.

39. InConcert. *InConcert Process Designer's Guide*. InConcert Inc., Cambridge, Massachusetts, 1997.

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

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

42. M. Klein, C. Dellarocas, and A. Bernstein, editors. *Towards Adaptive Workflow Systems, CSCW-98 Workshop, Proceedings*, Seattle, Washington, November 1998. http://ccs.mit.edu/klein/cscw98/.

43. T.M. Koulopoulos. *The Workflow Imperative*. Van Nostrand Reinhold, New York, 1995.

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

45. T.W. Malone, K. Crowston, J. Lee, B. Pentland et al. Tools for Inventing Organizations: Toward a Handbook for Organizational Processes. *Management Science*, 45(3):425–443, 1999.

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

47. A. Oberweis. *Modellierung und Ausführung von Workflows mit Petri-Netzen*. Teubner-Verlaggesellschaft, Germany, 1996. In German.

48. Pallas Athena. *Protos User Manual*. Pallas Athena BV, Plasmolen, The Netherlands, 1997.

49. Y. Perreault and T. Vlasic. *Implementing Baan IV*. Macmillan Computer Publishing, New York, 1998.

50. L. Pomello, G. Rozenberg, and C. Simone. A Survey of Equivalence Notions of Net Based Systems. In G. Rozenberg, editor, *Advances in Petri Nets 1992*, volume 609 of *Lecture Notes in Computer Science*, pages 410–472. Springer, Berlin, Germany, 1992.

51. Promatis. *Income Workflow User Manual*. Promatis GmbH, Karlsbad, Germany, 1998.

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

53. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, Germany, 1985.

54. A. Sheth. From Contemporary Workflow Process Automation to Adaptive and Dynamic Work Activity Coordination and Collaboration. In R. Wagner, editor, *Database and Expert Systems Applications, 8th. International Workshop, DEXA'97, Proceedings*, pages 24–27, Toulouse, France, September 1997. IEEE Computer Society Press, Los Alamitos, California, 1997.

55. A. Sheth, K. Kochut, and J. Miller. Large Scale Distributed Information Systems (LSDIS) laboratory, METEOR project page. http://lsdis.cs.uga.edu/proj/meteor/meteor.html.

56. Software-Ley. *COSA User Manual*. Software-Ley GmbH, Pullheim, Germany, 1998.

57. Staffware. *Staffware 97 / GWD User Manual*. Staffware Plc, Berkshire, UK, 1997.

58. H.M.W. Verbeek and W.M.P. van der Aalst. Woflan Home Page. http://www.win.tue.nl/˜woflan.

59. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. Computing Science Report 99/02, Eindhoven University of Technology, Eindhoven, The Netherlands, May 1999.

60. M. Voorhoeve and W.M.P. van der Aalst. Conservative Adaption of Workflow. In M. Wolf and U. Reimer, editors, *Practical Aspects of Knowledge Management (PAKM'96), 1st. International Conference, Workshop on Adaptive Workflow, Proceedings*, pages 1–15, Basel, Switzerland, October 1996. An extended version of the paper is available as Computing Science Report 96/24, Eindhoven University of Technology, Eindhoven, The Netherlands, 1996.

61. M. Voorhoeve and W.M.P. van der Aalst. Ad-hoc Workflow: Problems and Solutions. In R. Wagner, editor, *Database and Expert Systems Applications, 8th. International Workshop, DEXA'97, Proceedings*, pages 36–40, Toulouse, France, September 1997. IEEE Computer Society Press, Los Alamitos, California, 1997.

62. W.P. Weijland. *Synchrony and Asynchrony in Process Algebra*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 1989.

63. M. Wolf and U. Reimer, editors. *Practical Aspects of Knowledge Management (PAKM'96), 1st. International Conference, Workshop on Adaptive Workflow, Proceedings*, Basel, Switzerland, October 1996.