Adaptive Workflow: An Approach Based on Inheritance

W.M.P. van der Aalst[1,2] **T. Basten**[1] **H.M.W. Verbeek**[1]
P.A.C. Verkoulen[1,3] **M. Voorhoeve**[1]

[1] Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB, Eindhoven, The Netherlands.

[2] Large Scale Distributed Systems, Department of Computer Science, University of Georgia, Athens, GA 30602-7407, USA

[3] Origin DTS/Infrastructure Consultancy, Building VH-4, P.O. Box 6435, 5600 HK, Eindhoven, The Netherlands

Abstract

Today's information systems do not support adaptive workflow: either the information system abstracts from the workflow processes at hand and focuses on the management of data and the execution of individual tasks via applications or the workflow is supported by the information system but it is hard to handle changes. This paper addresses this problem by classifying the types of changes. Based on this classification, issues such as syntactic/semantic correctness, case transfer, and management information are discussed. It turns out that the trade-off between flexibility and support raises challenging questions. Some of these questions can be answered using advanced inheritance notions. This paper provides four inheritance-preserving transformation rules which can be used to avoid the typical problems related to change.

1 Introduction

Recently, many Workflow Management Systems (WFMSs) have become available. These systems signify that the term 'workflow management' is not just another buzzword. The phenomenon workflow management will have a large impact on the next generation of information systems. As the workflow paradigm continues to infiltrate organisations that need to cope with complex administrative processes, the workflow management system will become a fundamental building block [Koulopoulos, 1995; Jablonski and Bussler, 1996]. Therefore, the subject workflow management is of the utmost importance for people involved in the (re)design of administrative processes or the development of systems to support these processes.

At the moment, there are more than 200 workflow products commercially available [Lawrence, 1997] and many organisations are introducing workflow technology to support their business processes. It is widely recognised that workflow management systems should provide *flexibility* [Van der Aalst et al., 1998; Van der Aalst et al., 1999; Agostini and De Michelis, 1998; Casati et al., 1998; Ellis et al., 1995; Ellis et al., 1998; Han and Sheth, 1998; Heinl et al., 1998; Klein et al., 1998; Voorhoeve and Van der Aalst 1996; Wolf and Reimer 1996]. However, today's workflow management systems have problems dealing with *changes*, e.g., new technology, new laws, and new market requirements may lead to (structural) modifications of the workflow process definition at hand. In addition, ad-hoc changes may be necessary, e.g., because of exceptions. The inability to deal with various changes limits the application of today's workflow management systems.

Adaptive workflow aims at providing process support like normal workflow systems do, but in such a way that the system is able to deal with certain changes. These changes may range from ad-hoc changes such as changing the order of two tasks for an individual case (often called *exceptions*) to the redesign of a workflow process as the result of a Business Process Redesign (BPR) project.

Typical issues related to adaptive workflow are:

- *Correctness.* What kind of changes are allowed and is the resulting workflow process definition correct with respect to the criteria specified? We distinguish syntactic correctness (e.g., is it still a workflow?) and semantic correctness (e.g., can existing cases in the system be finished in a proper way?).
- *Dynamic change.* What to do with running instances (cases) of a workflow of which the definition has been changed? The term dynamic change refers to the problems that occur when running cases have to migrate from one process definition to another.
- *Management information.* How to provide a manager with aggregated information about the actual state of the workflow processes?

Taking these issues into account, a classification of the types of changes is presented. Based on this classification, potential problems are identified and pointers to solutions based on Petri-net theory are given. Moreover, four *inheritance-preserving transformation rules* are given. These rules can be used to facilitate the preservation of correctness (both syntactic and semantic), the circumvention of dynamic change anomalies, and the construction of aggregate management information.

2 Essence of workflow modeling

The term *Workflow Management* actually refers to the "logistics" of business processes. Workflow management does not focus on what information is being passed in a business process, but more on the control of the activity chain that is necessary to execute the business processes. The Workflow Management Coalition (WfMC) defines a workflow management system as follows: *"A system that completely defines, manages, and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic*" [Lawrence, 1997]. Workflows are *case-based*, i.e., every piece of work is executed for a specific case: an order, an insurance claim, a tax declaration, etc. The objective of a workflow management system is to handle these cases (by executing *tasks*) as efficiently and effectively as possible. The *workflow process definition* specifies which tasks need to be executed and in what order. When a task is executed for a case, this is usually done by using one or more *resources*, e.g., a machine, an employee, etc.

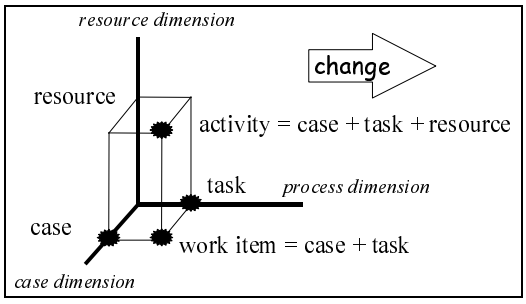

Figure 1. Three dimensions of workflow.

As can be seen in *Figure 1*, we identify three dimensions of a workflow [Van der Aalst, 1998b]:

- the *case dimension*, making clear that all cases are dealt with independently,
- the *process dimension*, specifying the overall workflow process, abstracting from individual cases,
- the *resource dimension*, depicting and classifying the resources used in the process.

In our work, we use Petri nets to represent workflow systems [Van der Aalst, 1998a; Van der Aalst, 1998b; Ellis and Nutt, 1993]. A Petri net represents a workflow if and only if it has exactly one starting place (*source*) and exactly one end place (*sink*) and the net obtained by adding a transition with the sink as the only input place and the source as the only output place is strongly connected. The latter is the case if for every two nodes x and y in the net, there is a path from x to y. Petri nets with this particular structure are called *workflow nets*. *Figure 5* shows two workflow nets. The left-hand-side process describes the sequential execution of four tasks (*A*, *B*, *C*, and *D*). Each task is modelled by a transition. Tasks are connected by places (represented by circles) to specify the ordering of tasks. Places may contain tokens (represented by black dots). The *state*, often called marking, is the distribution of tokens over places. A transition, i.e., a task, is enabled if and only if each of the input places contains a token. Enabled transitions can fire while removing tokens from the input places and putting tokens on the output places. In the sequential process shown in *Figure 5*, transition *C* is enabled because there is a token in the input place. Firing *C* will result in a situation where *D* is enabled. In the right-hand-side process in *Figure 5* tasks *B* and *C* are in parallel, i.e., they can be executed in any order. Note that *A* consumes one token and produces two tokens and that *D* consumes two tokens and produces one token. A detailed description of the class of workflow nets is beyond the scope of this paper and not needed for the remainder. However, some basic knowledge of Petri nets is needed to fully understand the concepts.

3 Classification of change

This section deals with the different kinds of changes and their consequences. Some of the perspectives relevant for change are:

- *process perspective*, e.g., tasks are added or deleted or their ordering is changed,
- *resource perspective*, e.g., resources are classified in a different way or new classes are introduced,
- *control perspective*, e.g., changing the way resources are allocated to processes and tasks,
- *task perspective*, e.g., upgrading or downgrading tasks,
- *system perspective*, e.g., changes to the infrastructure or the configuration of the engines in the enactment service.

In this paper, we restrict ourselves to changes in the process perspective as indicated in *Figure 1.*

Figure 2 shows that two kinds of change are identified:

- *Individual (ad-hoc) changes*, i.e., ad-hoc adaptation of the workflow process: a single case (or a limited set of cases) is affected. A good example is that of a hospital: if someone enters the hospital with a cardiac arrest, you are not going to ask him for his ID, although the workflow process may prescribe this. *Figure 2* distinguishes entry time changes (changes that occur when a case is not yet in the system) and on-the-fly changes (while in the system, the process definition for a case changes).

- *Structural (evolutionary) changes*, i.e., evolution of the workflow process: all new cases benefit from the adaptation. A structural change is typically the result of a BPR effort. An example of such a change is the change of a 4-year curriculum at a university to a 5-year one.

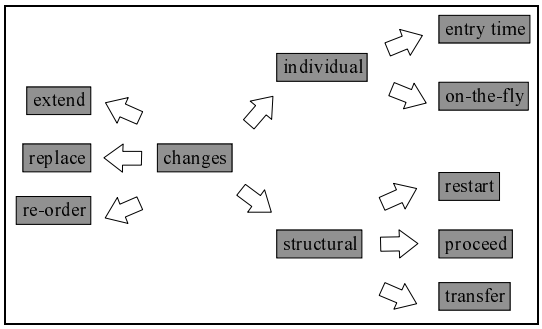

Figure 2. Classification of change.

On the left-hand side in *Figure 2*, we see the three different ways in which a workflow can be changed:

- the process definition is *extended* (e.g., by adding new tasks to cover process extensions),
- tasks are *replaced* by other tasks (e.g., a task is refined into a subprocess), and
- tasks in the process are *re-ordered* (e.g., two sequential tasks are put in parallel).

Figure 2 gives three possible alternatives for handling existing cases in the system when a process definition changes. Dealing with existing cases is only relevant in the case of a structural change because individual changes will always be (similar to) exceptions and, as such, will be dealt with by the one who initiated the change explicitly.

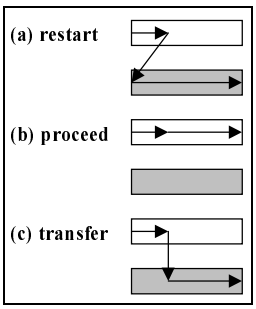

Figure 3. How to handle running cases?

Figure 3 shows the three alternatives: (a) *restart*: running cases are rolled back and restarted at the beginning of the new process, (b) *proceed*: changes do not affect running cases by allowing for multiple versions of the process, and (c) *transfer*: a case is transferred to the new process. The term *dynamic change* [Ellis et al., 1995] is used to refer to the latter policy.

4 Correctness

In the previous section, we presented some of the possibilities to change workflow specifications. In addition, we have presented some alternatives for dealing with cases that are somewhere in their workflow life-cycle the moment that the workflow is being changed.

These changes actually only make sense when they can be performed such that we can guarantee beforehand that the transformation satisfies a certain set of correctness-preservation properties. If we would not be able to make any statements about correctness preservation, it would mean that the new system would have no relationship with the old one. Being interested in adaptive workflow, this is obviously an undesired situation. Thus, it is very important that verification techniques and supporting tools are being developed that support such correctness-preserving transformations.

Two different kinds of correctness notions can be distinguished, viz., syntactic and semantic correctness.

S*yntactic* correctness is independent of the context, i.e., it refers to the minimal requirements any workflow should satisfy. For example, there should be no tasks without input places. Note that syntactic correctness not only refers to the structure of the workflow but also to the dynamic behaviour, e.g., potential deadlocks and livelocks are not allowed. A very important correctness criterion is the so-called *soundness property* that guarantees proper termination [Van der Aalst, 1998b]. Proper termination means that the state is reached with a single token in the sink place. *Figure 4* shows an incorrect workflow process definition. The execution of task *E* before task *B* will result in a deadlock because the case gets stuck in the state with just a token in *p5*. If task *C* is executed, then it is possible to execute *D* but a livelock is created because it is not possible to escape from the cycle formed by *B* and *F*. The workflow can be corrected by adding an arrow from *E* to *p2*, removing *p7*, and adding an arrow from *p4* to *F*. Today, Petri-net theory provides adequate tools to verify syntactic correctness. Nevertheless, such facilities are still missing in most of the commercial workflow management systems.

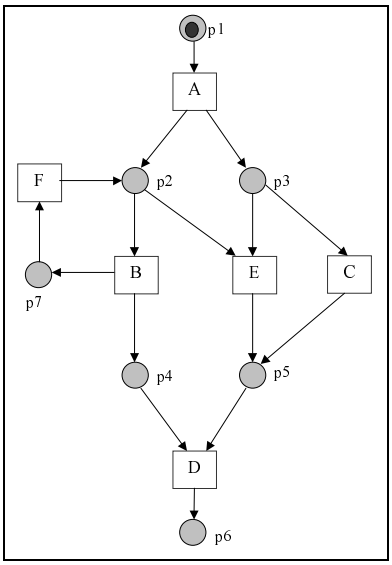

Figure 4. An incorrect workflow process definition.

Semantic correctness is concerned with the context in which the change occurs. Intuitively, semantic correctness deals with similarities between the capabilities of the old workflow and the capabilities of the new workflow, e.g., it may be desirable that the new workflow is able to handle cases the old way (but probably has some more functionality). Consider for example the two process definitions shown in *Figure 5*: the right-hand process can do more than the left-hand process but not vice versa. A similar relation holds between the right-hand process in *Figure 5* and the corrected version of the process shown in *Figure 4*. Comparing different variants of the same workflow requires advanced inheritance concepts [Basten, 1998; Van der Aalst and Basten, 1997; Voorhoeve and Van der Aalst, 1996]. In Section 7 four inheritance-preserving transformation rules are introduced. These rules can be used to *avoid* both syntactic and semantic errors.

5 Dynamic change

We identified three ways to deal with running cases after a structural process change: (a) restart, (b) proceed, and (c) transfer. Restarting cases causes no real difficulties except that it is often difficult to rollback the tasks that have already been executed. The proceed policy causes no problems. In fact, it is the only policy truly supported by today's commercial workflow management systems. The only policy that causes serious theoretical *and* practical problems is the transfer of cases. The term dynamic change refers to the problem of handling old cases in a new process, e.g., how to transfer cases to a new, i.e., improved, version of the process.

Figure 5 illustrates the dynamic change problem. If the sequential workflow process (left) is changed into the workflow process where tasks *B* and *C* can be executed in parallel (right) there are no problems, i.e., it is always possible to transfer a case from the left to the right. The sequential process has five possible states and each of these states corresponds to a state in the parallel process. For example, the state with a token in *s3* is mapped onto the state with a token in *p3* and *p4*. In both cases, tasks *A* and *B* have been executed and *C* and *D* still need to be executed.

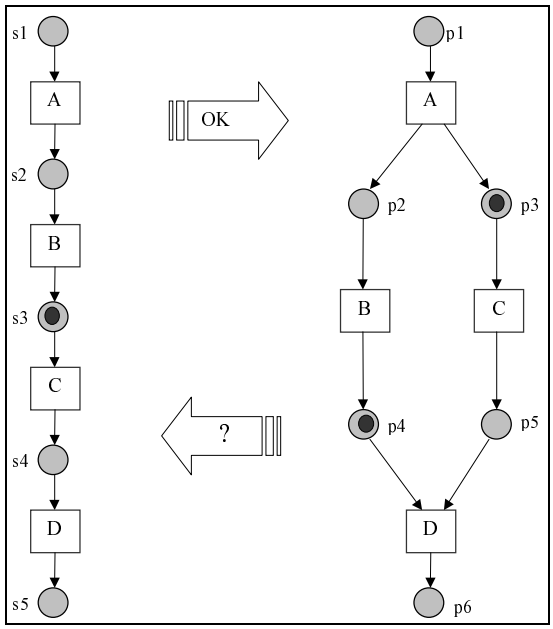

Figure 5. The dynamic change problem.

Now consider the situation where the parallel process is changed into the sequential one, i.e., a case is moved from the right-hand-side process to the left-hand-side process. For most of the states of the right-hand-side process this is no problem, e.g., a token in *p1* is moved to *s1*, a token in *p3* and a token in *p4* are mapped onto a single token in *s3*, and a token in *p4* and a token in *p5* are mapped onto a single token in *s4*. However, the state with a token in *p2* and *p5* (*A* and *C* have been executed) causes problems because there is no corresponding state in the sequential process (it is not possible to execute *C* before *B*). This simple example shows that it is not straightforward to migrate old cases to the new process after a change. Some authors have proposed a solution for the dynamic change problem [Agostini and De Michelis, 1998; Ellis et al., 1995; Ellis et al., 1998]. However, these solutions either require human intervention or are restricted to workflows with a particular structure. Note that both changes in *Figure 5* correspond to the reordering of tasks (see classification in Section 3).

6 Management information

Another problem of change is that it typically leads to multiple variants of the same process. For evolutionary

(i.e., structural) change the number of variants is limited. Ad-hoc changes may lead to the situation where the number of variants may be of the same order of magnitude as the number of cases. To manage a workflow process with different 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. *Figure 6* shows a workflow processes with two variants: a sequential one (left) and a parallel one (middle). The numbers in the places indicate the number of cases in a specific state, e.g., in the sequential process there are 3 cases in-between task *B* and task *C*, and in the parallel process there are 2 cases in-between *A* and *B*. Since the manager requires an aggregated view rather than a view for every variant of the workflow process, the cases need to be mapped onto a generalized version of the different processes. A solution is to find the 'greatest common divisor' (gcd) or the 'least common multiple' (lcm) for the two processes shown. Since all the states of the sequential process can be represented in the parallel process, we choose the parallel process to present management information. *Figure 6* shows the aggregated view of the two workflow processes (right). For all places in the right-hand-side process except *m3*, it is quite straightforward to verify that the numbers are correct. The number of tokens in place *m3* corresponds to the number of cases in-between *A* and *C*. In the sequential process, there are 1+3=4 cases in-between *A* and *C*. In the parallel process, there are also 4 cases in-between *A* and *C*, which brings the total to 8. For this small example, it may seem trivial to obtain this information. However, in general there are many variants of processes which may have up to 100 tasks and it is far from trivial to present aggregated information to the manager. The topic of generating management information was addressed in [Voorhoeve and Van der Aalst, 1996; Voorhoeve and Van der Aalst, 1997]. Despite its relevance for the next generation of workflow management systems only few researchers seem to be working on this topic.

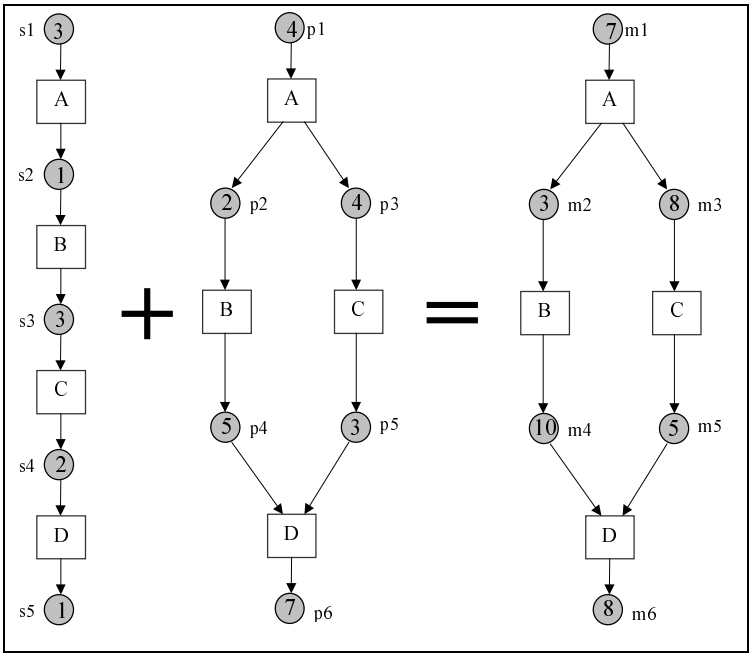

Figure 6. Mapping cases in different processes onto one workflow processes.

7 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 routing diagram) and *objects* (i.e., instances of the class) correspond to *cases*. In most object-oriented methods a class is characterized by a set of *attributes* and a set of *methods*. Attributes are used to describe properties of an object (i.e., an instance of the class). Methods symbolize operations on objects (e.g., create, destroy, and change attribute). The structure of a class is specified by the attributes and methods of that class. Note that the structure only refers to the static

aspects of the interface. The dynamic behavior of a class is either hidden inside the methods or modeled explicitly (in UML the life-cycle of a class is modeled in terms of state machines). Although the dynamic behavior is an intrinsic part of the class description (either explicit or implicit), inheritance of dynamic behavior is not well-understood. (See [Basten, 1998] for an elaborate discussion on this topic and pointers to related work.) Given the widespread use of inheritance concepts/mechanisms for the static aspects, this is remarkable. 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). Since workflow management aims at supporting business processes, these results are not very useful in this context. However, the work presented in [Basten, 1998; Van der Aalst and Basten, 1997] deals with inheritance of dynamic behavior in a comprehensive manner. This work is based on a particular class of Petri nets: the so-called sound workflow nets defined earlier. This class of Petri nets corresponds to workflow processes without deadlocks, livelocks, and other anomalies. Other inheritance-based approaches abstract from the causal relations between tasks/methods, i.e., the control or routing structure is not taken into account. 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 (InConcert, Inc, Cambridge, MA, USA) allows for building workflow class hierarchies. Unfortunately, inheritance is restricted to the attributes and the structure of the 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 [Basten, 1998; Van der Aalst and Basten, 1997] for the problems indicated in the previous two sections.

First we define four inheritance notions for workflow processes (i.e., workflows specified by workflow nets). Consider two workflow processes x and y. When is x a subclass of y? 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. In fact, the behavior with respect to the existing tasks should be identical.

In [Basten, 1998; Van der Aalst and Basten, 1997] we have identified four different notions of inheritance: *protocol inheritance*, *projection inheritance*, *protocol/projection inheritance*, and *life-cycle inheritance*. Protocol/projection inheritance is the most restrictive form of inheritance. If x is a subclass of y with respect to protocol/projection inheritance, then x is a subclass of y with respect to protocol inheritance *and* projection inheritance. Life-cycle inheritance is the most liberal form of inheritance, i.e., protocol and/or projection inheritance implies life-cycle inheritance.

The notion of projection inheritance is based on *abstraction*: *If it is not possible to distinguish 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 with respect to projection inheritance.*

For distinguishing x and y under projection inheritance we only consider the tasks present in both nets (i.e., in y). All other tasks in x are renamed to τ. One can think of these tasks as silent, internal, or not observable. Since *branching bisimulation* [Basten, 1998] is used as an equivalence notion, we abstract from transitions with a τ label, i.e., for deciding whether x is a subclass of y only the tasks with a label different from τ are considered. The behavior with respect to these tasks is called the *observable behavior*. Added tasks (i.e., tasks present in x but not in y) can be executed but are not observable by the outside world, i.e., projection inheritance conforms to *hiding* or *abstracting* from tasks new in x.

The notion of protocol inheritance is based on *encapsulation* (i.e., blocking of new tasks): *If it is not possible to distinguish x and y when only tasks of x that are also present in y are executed, then x is a subclass of y.*

For distinguishing x and y under protocol inheritance all tasks present in x but not in y are blocked. The new tasks are simply disallowed to be executed.

A formal definition of the four forms of inheritance is beyond the scope of this paper. (The definition builds on branching bisimulation equivalence and an abstraction operator which renames a given set of tasks to τ.) The interested reader is referred to [Basten, 1998; Van der Aalst and Basten, 1997].

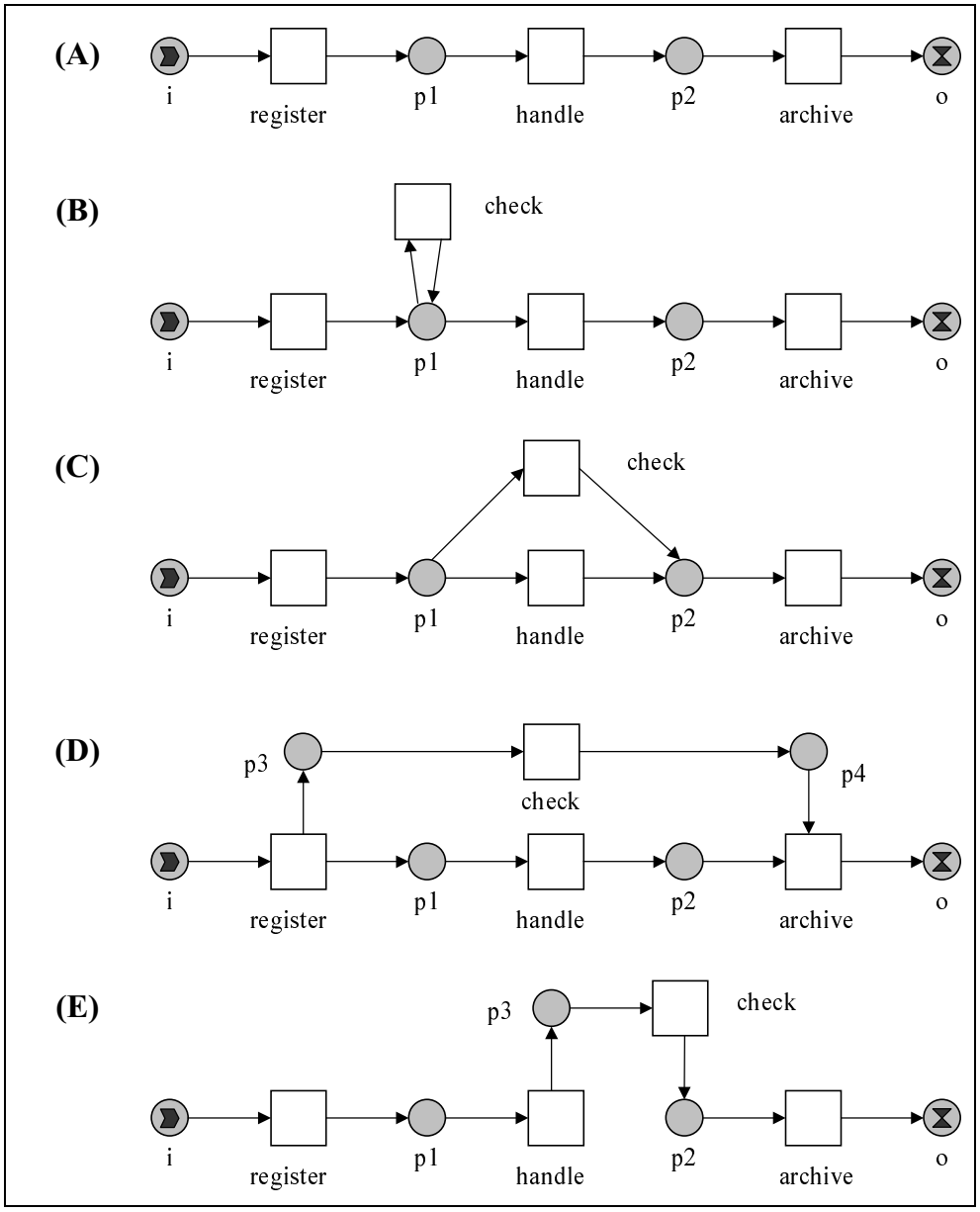

Figure 7. Five routing diagrams describing variants of a simple workflow process.

Figure 7 shows five workflow processes modeled in terms of workflow nets. Special symbols are used to indicate the source place (i) and the sink place (o). Workflow process (A) consists of three sequential tasks: *register*, *handle*, and *archive*. Each of the other workflow processes extends this process with one additional task: *check*. In workflow process (B) task *check* can be executed arbitrarily many times between *register* and *handle*. Workflow process (B) is a subclass of workflow process (A) with respect to projection inheritance; if task *check* is abstracted from, then the two processes behave equivalently (i.e., are branching bisimilar). Workflow process (B) is also a subclass of workflow process (A) with respect to protocol inheritance; blocking task *check* yields two equivalent processes. Workflow process (C) is not a subclass with respect to projection inheritance; hiding task *check* introduces the possibility to skip task *handle* and thus change the actual behavior. However, (C) is a subclass of (A) with respect to protocol inheritance. Workflow process (D) is a subclass of workflow process (A) with respect to projection inheritance; hiding this task results in two equivalent processes. However, (D) is not a subclass of (A) with respect to protocol inheritance; blocking task *check* results in a deadlock. Workflow process (E) is a subclass of workflow process (A) with respect to projection inheritance; the detour via task *check* can be hidden thus yielding an observable behavior identical to (A). Workflow process (E) is not a subclass of workflow process (A) with respect to protocol inheritance. All workflow processes are a subclass of (A) with respect to

life-cycle inheritance. For life-cycle inheritance some of the new tasks are blocked and others are hidden to obtain two equivalent processes. Only workflow process (B) is a subclass with respect to protocol/projection inheritance and both (B) and (C) are subclasses of (A) with respect to protocol inheritance.

In [Basten, 1998; Van der Aalst and Basten, 1997] we proposed a number inheritance-preserving transformation rules. These rules correspond to frequently used design constructs and preserve one or more of the four inheritance notions. A detailed description of these rules is beyond the scope of this paper. Therefore, we give an informal description of four inheritance rules: PP, PT, PJ, and PJ3.

Protocol/projection inheritance-preserving transformation rule PP is illustrated by *Figure 8*. New transitions (i.e., tasks) and places (i.e., conditions) are added to the original workflow net such that tokens are only temporally removed from a place in the original net. The added subnet may have any structure as long as it is guaranteed that any token taken from place p will be returned eventually and no tokens are left in the subnet. Since the subnet only postpones behavior, the extended workflow net (right) is a subclass of the original workflow net (left) under both projection and protocol inheritance. If one abstracts from the newly added tasks or blocks the newly added tasks, one cannot find any differences between both nets.

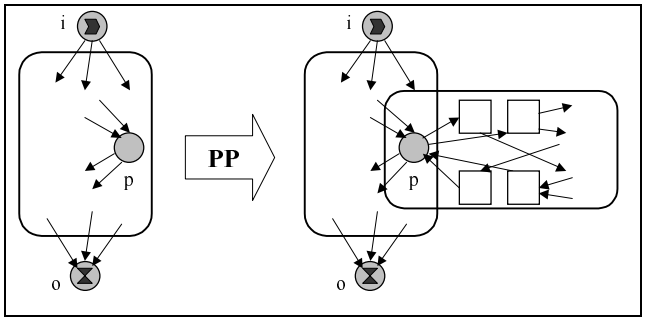

Figure 8. Inheritance-preserving transformation rule PP.

Figure 9 illustrates transformation rule PT. PT preserves protocol inheritance and adds alternative behavior. The added subnet removes tokens from the original net to execute tasks not present in the original net. The added subnet may have any structure as long as any token taken from place p_i will be returned in place p_o eventually and no tokens are left in the subnet. Other requirements are that all new tasks consuming tokens from p_i should not appear in the original net and that the routes via the subnet do not create new states in the original net. It is easy to see that PT preserves protocol inheritance; the added subnet is never activated if all new tasks are blocked. Note that PT does not preserve projection inheritance since the subnet can be used to bypass parts of the original net.

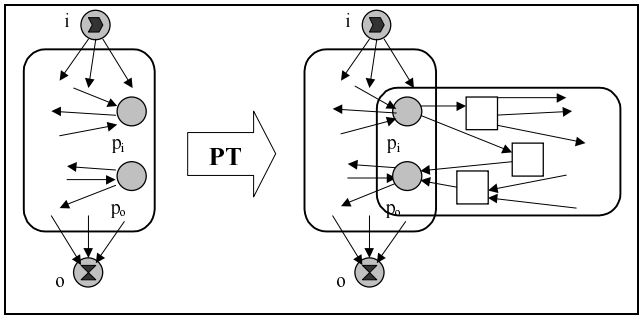

Figure 9. Inheritance-preserving transformation rule PT.

Figure 10 shows inheritance-preserving transformation rule PJ. Rule PJ inserts new tasks in-between a task t_p and a place p in the original workflow net. In fact, rule PJ can be used to insert an arbitrary subflow in-between t_p and p. The added subnet may have any structure as long as it is guaranteed that once the subnet is activated by firing t_p eventually a token is put in place p and no tokens are left behind. It is easy to see that the extended workflow net (right) is a subclass of the original workflow net (left) under projection inheritance: by abstracting from the newly added tasks the observable behaviors coincide.

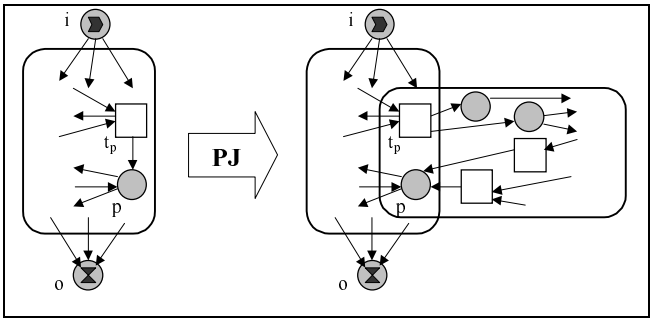

Figure 10. Inheritance-preserving transformation rule PJ.

Projection inheritance-preserving transformation rule PJ3 can be used to add parallel behavior. (The rule is named PJ3 for historical reasons.) *Figure 11* illustrates this rule. The execution of task t_i activates the subnet containing new tasks to be executed in parallel. Task t_o synchronizes the original workflow net and the added subnet. The added subnet may use arbitrary routing constructs as long as (1) the execution of t_i is always followed by the execution of t_o in the original net and t_o is always preceded by t_i, (2) activation of the subnet via firing t_i is always followed by a state which marks the input places of t_o in the subnet, and (3) no tokens are left behind in the subnet after firing t_o. If these three requirements are guaranteed, then the extended workflow net (right) is a subclass of the original workflow net (left) under projection inheritance.

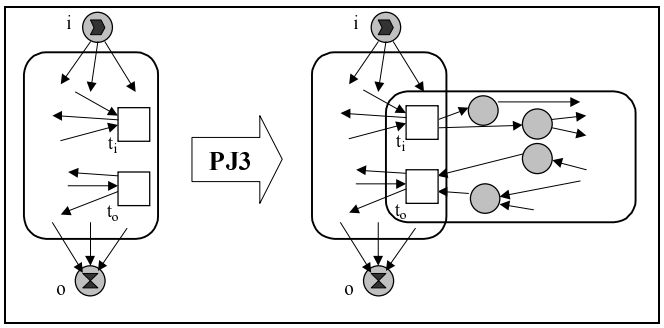

Figure 11. Inheritance-preserving transformation rule PJ3.

For a formal definition of these rules we refer to [Basten, 1998; Van der Aalst and Basten, 1997]. Details and subtle requirements are omitted to simplify the presentation of the main ideas. The workflow nets shown in *Figure 7* illustrate the four rules. Rule PP introduces new tasks which only postpone behavior. Workflow process (B) shown in *Figure 7* can be constructed from (A) by applying this rule; task *check* only postpones the execution of *handle*. Rule PT introduces alternative behavior. Workflow process (C) shown in *Figure 7* can be constructed from (A) by applying PT. Rule PJ inserts new tasks in-between existing tasks. Workflow process (A) shown in *Figure 7* can be extended to workflow process (E) using this rule. The extension can be a single task but also a complex subflow containing many tasks and all kinds of causality relations. Rule PJ3 adds parallel behavior. Workflow process (A) shown in *Figure 7* can be extended to workflow process (D) using this rule. The rules correspond to design constructs that are often used in practice, namely iteration, sequential composition, and parallel composition. If the designer sticks to these rules, inheritance is guaranteed!

The inheritance-preserving rules can be used to *avoid* the problems indicated in this paper. First of all, the four rules enable the designer to establish syntactic and semantic correctness in a compositional manner. For example, soundness can be verified by analyzing the original part and the extension separately. Second, if the designer sticks to the inheritance-preserving transformation rules, then it is possible to guarantee instant transfers without the anomalies described in Section 5. Suppose that x is a subclass of y constructed using the rules PP, PT, PJ, and PJ3. For any state in workflow process y it is possible to transfer a case to x such that the transfer is instantaneous (i.e., no postponements needed) and does not introduce syntactic errors (e.g., deadlocks, livelocks, and improper termination) nor semantic errors (e.g., the double execution of tasks or unnecessary skipping of tasks). Moreover, it is also possible to transfer cases from subclass x to superclass y without any problems. Finally, the inheritance-preserving transformation rules can be used to construct aggregate management information automatically. States of the subclass can be mapped onto states of the superclass and vice versa. Therefore, all cases can be mapped onto one workflow process if all variants/versions are related via the rules PP, PT, PJ, and PJ3. Moreover, given a set of variants/versions of some process, it is possible to define 'greatest common divisor' (gcd) or the 'least common multiple' (lcm) using the inheritance notions and thus create appropriate management views. A detailed discussion and formal proofs of these statements are beyond the scope of this paper. The interested reader is referred to [Van der Aalst and Basten, 1999] for examples, formal definitions and proofs.

8 Conclusion

In this paper, we discussed some of the problems that need to be solved to enable adaptive workflow. Most of the problems stem from the fact that flexibility (the ability to handle changes) on the one hand and process support (enactment, control, and management information) on the other hand impose (partially) conflicting constraints. We have classified the various forms of change. Based on this classification we pointed out some solutions for the problems identified. Future work in this area will focus on verification (semantic correctness), relating different versions of a workflow process for supporting dynamic change, and generating management information.

This paper extends the results presented in [Van der Aalst et al., 1999] with an approach based on inheritance. It is our belief that the inheritance-preserving transformation rules presented in Section 7 are a good starting point for solving the problems identified. Each of the rules corresponds to a design construct which is often used in practice, namely choice, parallel composition, sequential composition, and iteration. The rules preserve to some extent syntactic and semantic correctness. Future research will focus on the application of inheritance notions to deal with dynamic change and the generation of useful management information.

References

[Van der Aalst, 1998a] W.M.P. van der Aalst. Chapter 10: Three Good reasons for Using a Petri-net-based Workflow Management System. In T. Wakayama et al., editors, Information and Process Integration in Enterprises: Rethinking documents, Kluwer Academic Publishers, 1998.

[Van der Aalst, 1998b] 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.

[Van der Aalst and Basten, 1997] 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, volume 1248 of Lecture Notes in Computer Science, pages 62-81. Springer, Berlin, Germany, 1997.

[Van der Aalst et al., 1998] W.M.P. van der Aalst, G. De Michelis, and C.A. Ellis (editors). Proceedings of Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM'98). Computing Science Report 98/7, Eindhoven University of Technology, Eindhoven, the Netherlands, 1998.

[Van der Aalst et al., 1999] W.M.P. van der Aalst, T. Basten, H. Verbeek, P. Verkoulen, and M. Voorhoeve. Adaptive Workflow: On the Interplay between Flexibility and Support. In J. Filipe and J. Cordeiro (editors), Proceedings of the first International Conference on Enterprise Information Systems, pages 353-360, Setubal, Portugal, 1999.

[Van der Aalst and Basten, 1999] W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach Tackling the Problems Related to Change. Technical report, 1999.

[Agostini and De Michelis, 1998]. A. Agostini and G. De Michelis. Simple Workflow Models. In [Van der Aalst et al., 1998], pages 146-164.

[Basten, 1998] T. Basten. In Terms of Nets: System Design with Petri Nets and Process Algebra. PhD Thesis. Eindhoven University of Technology, Department of Computing Science, Eindhoven, the Netherlands, 1998.

[Casati et al., 1998] F. Casati, C. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. Data and Knowledge Engineering, 24(3):211-238, 1998.

[Ellis et al., 1995] C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In N. Comstock and C.A. Ellis, editors, Conf. on Organizational Computing Systems, pages 10 - 21. ACM SIGOIS, ACM. Milpitas, California, 1995.

[Ellis et al., 1998] C.A. Ellis, K. Keddara, and J. Wainer. Modeling Workflow Dynamic Changes Using Timed Hybrid Flow Nets. In [Van der Aalst et al., 1998], pages 109-128.

[Ellis and Nutt, 1993] 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, volume 691 of Lecture Notes in Computer Science, pages 1-16. Springer, Berlin, Germany, 1993.

[Han and Sheth, 1998] Y. Han and A. Sheth. On Adaptive Workflow Modeling. In Proceedings of the 4th International Conference on Information Systems Analysis and Synthesis, pages 108-116, Orlando, Florida, 1998

[Heinl et al. 1998] P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein, and M. Teschke. A comprehensive approach to flexibility in workflow management systems. Technical report TR-16-1998-6, University of Erlangen-Nuremberg, Erlangen, Germany, 1998.

[Jablonski and Bussler, 1996] S. Jablonski and C. Bussler. Workflow Management: Modeling Concepts, Architecture, and Implementation International Thomson Computer Press, 1996.

[Klein et al., 1998] M. Klein, C. Dellarocas, and A. Bernstein, (editors). Proceedings of the CSCW-98 Workshop Towards Adaptive Workflow Systems, Seattle, 1998.

[Koulopoulos, 1995] T.M. Koulopoulos. The Workflow Imperative. Van Nostrand Reinhold, New York, 1995.

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

[Sheth, 1997] A. Sheth. From Contemporary Workflow Process Automation to Dynamic Work Activity Coordination and Collaboration. Siggroup Bulletin, 18(3):17-20, 1997.

[Voorhoeve and Van der Aalst, 1996] M. Voorhoeve and W.M.P. van der Aalst. Conservative Adaption of Workflow. In [Wolf and Reimer, 1996], pages 1-15.

[Voorhoeve and Van der Aalst, 1997] M. Voorhoeve and W.M.P. van der Aalst. Ad-hoc Workflow: Problems and Solutions. In R. Wagner, editor, Proceedings of the 8th DEXA Conference on Database and Expert Systems Applications, pages 36-41, Toulouse, France, 1997.

[Wolf and Reimer, 1996]. M. Wolf and U. Reimer (editors). Proceedings of the International Conference on Practical Aspects of Knowledge Management (PAKM'96), Workshop on Adaptive Workflow, Basel, Switzerland, 1996.