How to handle dynamic change and capture management information?
An approach based on generic workflow models

W.M.P. van der Aalst[†]
Department of Information and Technology
Faculty of Technology Management
Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
w.m.p.v.d.aalst@tm.tue.nl

Abstract
Today's workflow management systems have problems dealing with both ad-hoc changes and evolutionary changes. As a result, the workflow management system is not used to support dynamically changing workflow processes or the workflow process is supported in a rigid manner, i.e., changes are not allowed or handled outside of the workflow management system. This paper addresses two notorious problems related to adaptive workflow: (1) providing *management information* at the right aggregation level, and (2) supporting *dynamic change*, i.e., migrating cases from an old to a new workflow. These two problems are tackled by using generic process models. A generic process model describes a family of variants of the same workflow process. To relate members of a family of workflow processes we propose notions of *inheritance.* These notions of inheritance are used to address the two problems mentioned both a design-time and at run-time.

Keywords
Workflow management, adaptive workflow, management information, inheritance, dynamic change, generic product models.

1 Introduction
Workflow management promises a new solution to an age-old problem: controlling, monitoring, optimizing and supporting business processes [38,39,51]. What is new about workflow management is the explicit representation of the business process logic which allows for computerized support. At the moment, there are more than 200 workflow products commercially available and many organizations are introducing workflow technology to support their business processes. A critical challenge for workflow management systems is their ability to respond effectively to changes [11,12,13,17,20,21,27,30,37,41,43,49]. Changes may range from ad-hoc modifications of the process for a single customer to a complete restructuring for the workflow process to improve efficiency. Today's workflow management systems are ill suited to dealing with change. They typically support a more or less idealized version of the preferred process. However, the real run-time process is often much more variable than the process specified at design-time. The only way to handle

[†] Part of this work was done at AIFB (University of Karlsruhe, Germany), LSDIS (University of Georgia, USA), CTRG (University of Colorado, USA) during a sabbatical leave.

changes is to go behind the system's back. If users are forced to bypass the workflow management system quite frequently, the system is more a liability than an asset. Therefore, we take up the challenge to find techniques to add flexibility without loosing the support provided by today's systems.

Typically, there are two types of changes: (1) *ad-hoc changes* and (2) *evolutionary changes*. Ad-hoc changes are handled on a case-by-case basis. In order to provide customer specific solutions or to handle rare events, the process is adapted for a single case or a limited group of cases. Evolutionary change is often the result of reengineering efforts. The process is changed to improve responsiveness to the customer or to improve the efficiency (do more with less). The trend is towards an increasingly dynamic situation where both ad-hoc and evolutionary changes are needed to improve customer service and reduce costs.

This paper presents an approach to tackle the problem of change. This approach is inspired by the techniques used in *product configuration* [46]. As factories have to manufacture more and more customer specific products, the trend is to have a very high number of variants for one product. Products, like a car or a computer, can have millions of variants (e.g., combinations of color, engine, transmission, and options). Also product specifications and their components evolve at an increasing pace. Product configuration deals with these problems and has been a lively area of research for the last decade. Moreover, some solutions have already been implemented in today's enterprise resource planning systems such as SAP and Baan. To deal with changes the traditional *Bill-Of-Material* (BOM) is extended with product families. A product family corresponds to a range of product types and allows for the modeling of generic product structures. The term *generic BOM* [23,29,46,47] is used when generic product structures are described by means of an extension to the traditional BOM. In this paper, we extend traditional process modeling techniques in a similar manner. We adopt the notion of *process families* to construct *generic workflow process models*.

A generic workflow process model is a process model which can be configured to accommodate flexibility and enables both ad-hoc and evolutionary changes. Using generic workflow process models, the workflow management system can support the design and enactment (i.e., execution) of processes subject to change. Moreover, the generic process model introduced in this paper allows for the navigation through two dimensions: (1) the vertical dimension (is-part-of/contains) and (2) the horizontal dimension (generalizes/specializes). Although the second dimension is absent in today's workflow management systems, it is of the utmost importance for the reusability and adaptability of workflow processes.

The addition of the horizontal dimension allows for the design and enactment of many variants of a workflow process. However, it is not sufficient to support the design and enactment. There are two additional issues that need to be dealt with: (1) *management information* [49,50], and (2) *dynamic change* [13,20,21]. In spite of the existence of many variants of one process, the manager is interested in information at an aggregate level, i.e., management information which abstracts from small variations. 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.

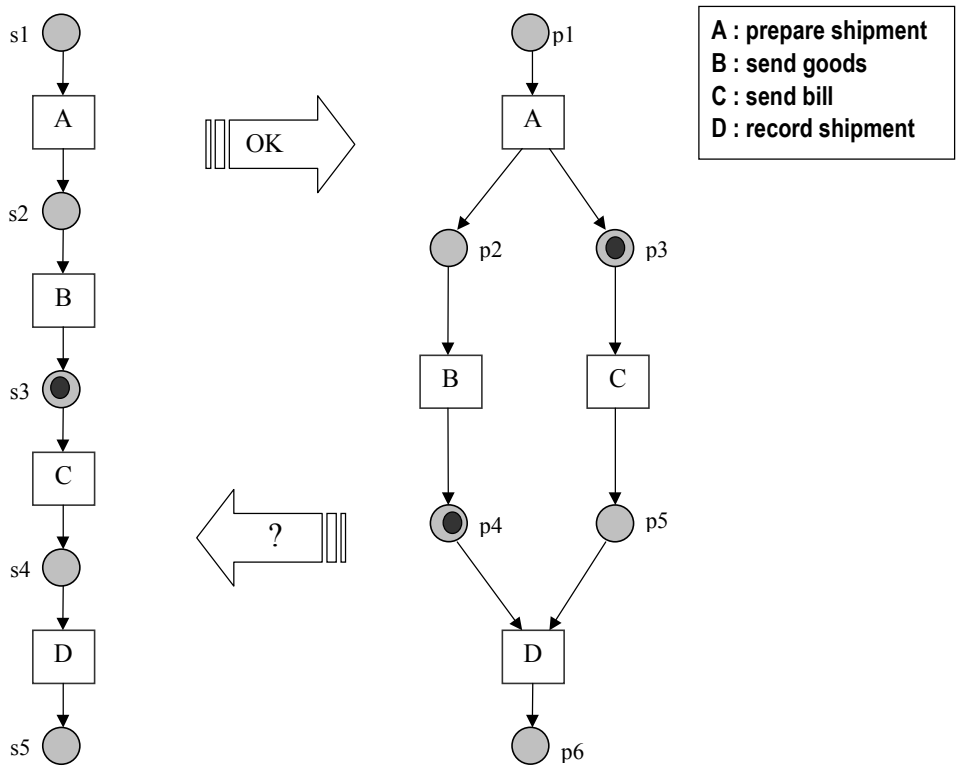

Figure 1: The dynamic change problem.

Figure 1 illustrates the dynamic change problem[1]. The left-hand-side process executes the tasks *prepare shipment*, *send goods*, *send bill*, and *record shipment* in sequential order. In the right-hand-side process the sending of the goods and the sending of the bill can be executed in parallel, i.e., there is no ordering relation between the tasks *send goods* and *send bill*. In the remainder we will use identifiers A, B, C, and D to denote the four tasks. 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. 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 *p4* are mapped onto one token in *s3*, and a token in *p4* and a token *p5* are mapped onto one token in *s4*. However, the state with a token in both *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*). The example in Figure 1 shows that it is not straightforward to migrate old cases to the new process after a change.

[1] In this paper, we use Petri nets to illustrate the main concepts. Readers not familiar with Petri nets are referred to the appendix.

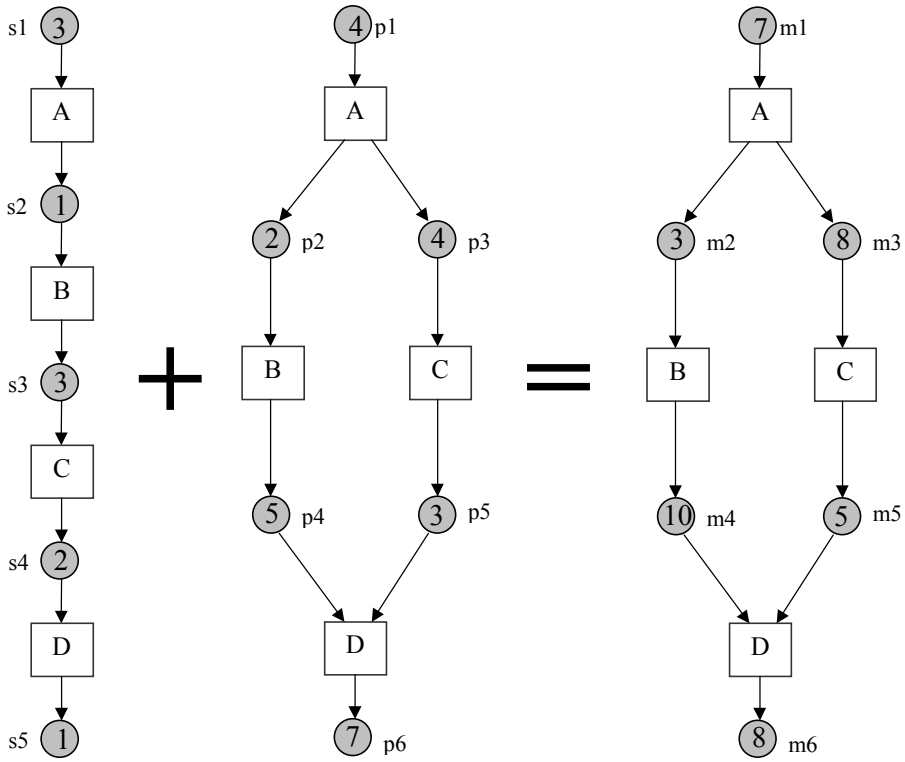

Figure 2: Aggregated management information.

Another problem of change is that it typically leads to multiple variants of the same process. For evolutionary change the number of variants is limited. 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. 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 2 shows a workflow processes with two variants: a sequential one (left) and a parallel one (middle). The numbers 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. Therefore we need to find the 'greatest common denominator' or the 'least common multiple' for the two processes shown. Since all the states of the sequential process are presented in the parallel process, we choose the parallel process to present the management information. Figure 2 shows the aggregated view of the two workflow processes (right). For all places in the right-hand-side process except *m3*, is quit 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 and the processes may have up to 100 tasks and it is far from trivial to present aggregated information to the manager.

These two issues (dynamic change and management information) cause a lot of problems which need to be solved. We think that it is possible to tackle these problems by using the notion of a *minimal representative* of a generic process. By mapping states on this minimal representative it may be possible to generate adequate management information. Moreover, linking states of the members of a process family to the states of a minimal representative seems to be useful for the automated support of dynamic change. To support the construction of the minimal representative and the mapping of cases from members of a process family to states of a minimal representative and vice versa, we propose an approach based on the *inheritance preserving transformation rules* presented in [6,14].

This paper is organized as follows. First we classify the types of changes that we would like to support. Then, we introduce an approach to specify generic process models using two types of diagrams: routing diagrams and inheritance diagrams. It is shown that this approach facilitates dealing with all kinds of changes. In the second part of the paper, we show that the notion of a minimal representative of a generic process can be used to tackle the problems involving dynamic change and management information. To make the approach more concrete, we introduce four inheritance preserving transformation rules that can be used as a carrier for dynamically transferring cases and obtaining management information.

2 Adaptive workflow

Workflows are typically *case-based*, i.e., every piece of work is executed for a specific *case*. Examples of cases are a mortgage, an insurance claim, a tax declaration, an order, or a request for information. Cases are often generated by an external customer. However, it is also possible that a case is generated by another department within the same organization (internal customer). The goal of workflow management is to handle cases as efficient and effective as possible. A workflow process is designed to handle similar cases. Cases are handled by executing *tasks* in a specific order. The routing definition specifies which tasks need to be executed and in what order. Alternative terms for routing definition are: ‘procedure’, ‘flow diagram’ and ‘workflow process definition’. In the routing definition, routing elements are used to describe sequential, conditional, parallel and iterative routing thus specifying the appropriate route of a case (WfMC [39,51]). 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: 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, or employee). In office environments where workflow management systems are typically used, 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 a workflow, we see cases, work items and activities. Work items link cases and tasks. Activities link cases, tasks, and resources. See [1,3,4,9,10,17,22,33,39,42,51] for more information about workflow concepts and the modeling of workflow processes.

Adaptive workflow is an area of research which examines concepts, techniques, and tools to support change. It is widely recognized that workflow management systems should provide *flexibility* [13,17,20,21,27,30,43,49]. However, as indicated in the introduction, today's workflow management systems have problems dealing with change. New technology, new laws, and new market requirements lead to modifications of the workflow process definitions at hand. Last minute changes on a case-by-case basis lead to all kinds of exceptions. The inability to deal with various changes limits the application of today's workflow management systems. The limitations of today's workflow management systems and current approaches with respect to flexibility raise a number of interesting questions. In fact, several workshops have been organized to discuss the problems related to workflow change [12,36,52]. In this paper we restrict ourselves to changes with respect to the routing of cases, i.e., the control flow. We abstract from organizational changes, i.e., we do not consider adaptations of the resource classification and the mapping of work items onto resources. We also abstract from the contents of tasks.

The restriction to consider only the routing definition allows us to classify changes as follows [8]:

- *Ad-hoc change*
  Changes occurring on an individual basis: only a single case (or a limited set of case) is affected. 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 ad-hoc change is skipping a task in case of an emergency. This kind of change is often initiated by some external cause. A typical dilemma is to decide what kinds of changes are allowed. Another problem related to ad-hoc change is the fact that it is impossible to foresee all possible changes. For ad-hoc change we distinguish between the moment of change:
  - *Entry time*
    The routing definition is frozen the moment the processing of the case starts, i.e., no changes are allowed during the processing.
  - *On-the-fly*
    Changes are allowed at any moment, i.e., the process may change while the case is being handled. Ad-hoc on-the-fly changes allow for self-modifying routing definitions.
- *Evolutionary change*
  Changes of a structural nature: from a certain moment in time, the process 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 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. We identify three ways to deal with existing cases:

- *Restart*
  All existing cases are aborted and restarted. 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.
- *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.
- *Transfer*
  Existing cases are transferred to the new process, i.e., they can directly benefit from evolutionary changes. The term *dynamic change* is used to refer to the problem of transferring cases to a consistent state in the new process.

Both for ad-hoc and evolutionary change, we distinguish three ways in which the routing of cases along tasks can be changed:

- *Extend*
  Adding new tasks which (1) are executed in parallel, (2) offer new alternatives, or (3) are executed in-between existing tasks.
- *Replace*
  A task is replaced by another task or a subprocess (i.e., refinement), or a complete region is replaced by another region.
- *Re-order*
  Changing the order in which tasks are executed without adding new tasks, e.g., swapping tasks or making a process more or less parallel.

This concludes our classification of workflow change. Note that the term *exception handling* does not appear in the classification [19,26,36,44]. An exception is the occurrence of some unexpected or abnormal event. In most cases, exceptions are undesirable because they generate additional complications and work. If a workflow management system provides an exception handler, it is possible to specify the actions to be performed in order to respond to certain exceptions. However, often the humans participating in the process are the "real" exception handlers, because it is not possible to pre-specify all possible exceptions. Note that an exception is not a change. Exceptions only trigger changes. Exceptions generated by external actors (e.g. a customer reporting an emergency) typically lead to ad-hoc changes. Exceptions generated by internal actors (e.g. the breakdown of an information system) typically lead to the blocking of parts of the workflow or to (temporary) evolutionary changes.

It is interesting to compare the classification shown in **Error! Reference source not found.** with the classification for failures and exceptions given in [18]. In [18], Eder and Liebhart, describe four types of failures and exceptions:

- *Basic failures*
  Failures such as a system crash, connection problems, or the breakdown of the underlying database system.
- *Application failures*
  Failures of an application program launched or managed by the workflow management system. These errors are typically the result of unexpected input.

- *Expected exceptions*
  Workflow executions that do not correspond to the "normal" behavior but still occur frequent enough to be anticipated, e.g., a customer does not return a form. The handling of these exceptions can be specified at design time but, if mixed with the normal flow, typically results in "spaghetti-like" models.
- *Unexpected exceptions*
  Exceptions that are so rare that they cannot be anticipated and therefore need to be handled at run-time.

The first two types are handled at the system or application level and typically do not result in process changes. The latter two types of failures and exceptions, i.e., expected exceptions and unexpected exceptions, may generate process changes. Unexpected exceptions typically generate on-the-fly ad-hoc changes. Expected exceptions can be modeled explicitly thus avoiding any changes at run-time. However, many expected exceptions are handled in an ad-hoc manner to simplify the process model.

The classification shown in **Error! Reference source not found.** reveals that there are many types of changes causing different types of problems. Typically, changes lead to many variants of the same process. Therefore, a lot of routing definitions need to be stored and supported by the workflow enactment service. To keep track of these definitions and to avoid redundancy they should be stored in a structured way. Having many variants emphasizes the fact that it is important to support automatic verification: given a set of criteria, all changes should be checked before the routing definition is put into production. Moreover, it is important to be able to provide the manager with aggregated information and support dynamic change. To solve some of these problems, we propose an approach which allows for the formulation of generic process models.

3 Generic process models

A generic process model is specified by a set of routing diagrams and inheritance diagrams. Before these two diagram types are presented, we introduce the basic concepts and the relations between these concepts.

3.1 Concepts

Cases are the objects which need to be handled by the workflow (management system). The Workflow Management Coalition (WfMC) uses the term "process instance" to denote a case [39,51]. Examples of cases are tax declarations, complaints, job applications, credit card payments, and insurance claims. A *task*, also referred to as "activity" by the WfMC [39,51], is an atomic piece of work. A task is concrete, i.e., it can be specified, but is not specific for a single case. In principle, a task can be executed for any case. A *non-atomic concrete process* is similar to a task but it is not atomic. A non-atomic concrete process is specified by a routing diagram and corresponds to a case type rather than a specific case. The WfMC [39,51] uses the terms "process" and "subprocess" to refer to such diagrams. Figure 4 and Figure 5 show non-atomic concrete processes modeled with COSA respectively Staffware. A *concrete process* is either a task or a non-atomic concrete process, i.e., it is a pre-specified piece of work which can be executed for many cases (if needed). A *generic process* is not specified, i.e., it is not concrete but refers to a family of processes. Since it is not concrete, it makes no sense to distinguish between atomic and non-

atomic generic processes. In fact, one generic process may signify both concrete tasks and non-atomic concrete processes at the same time. One can think of a generic process as a placeholder for a concrete process. For example a process called *procurement* may contain the generic process *contact_supplier*. This generic process is not modeled in terms or a routing diagram but refers to the tasks *phone_supplier*, *fax_supplier*, and *e-mail_supplier*, i.e., *contact_supplier* is only a placeholder: The actual processing is handled as specified by one of these tasks. Note that a generic process may also refer to non-atomic concrete processes and even other generic processes. A *process node* is either a concrete process or a generic process, i.e., *procurement, contact_supplier, phone_supplier*, *fax_supplier*, and *e-mail_supplier* are all process nodes. A routing diagram contains process nodes, i.e., a non-atomic concrete process is specified in terms of both concrete and generic processes. A process node appears in zero or more routing diagrams. In each routing diagram, process nodes are connected by *routing elements* specifying the order in which the process nodes need to be executed. A process node refers to zero or more generic processes. If a process node X refers to a generic process Y, then X belongs to the process family described by Y and we say that X *is a child of* Y. A concrete process can be the child of a generic process, a generic process can be the child of another generic process, but a generic process cannot be the child of a concrete process. Note that a process node can be the child of many generic processes. Each case refers to precisely one non-atomic concrete process. Since the routing diagram describing a non-atomic concrete process may contain generic processes, it is necessary to instantiate generic processes by concrete processes for specific cases, i.e., for a specific case, generic processes in the routing diagram are replaced by concrete processes. Consider for example the procurement process where the generic process *contact_supplier* is instantiated by the task *phone_supplier*.

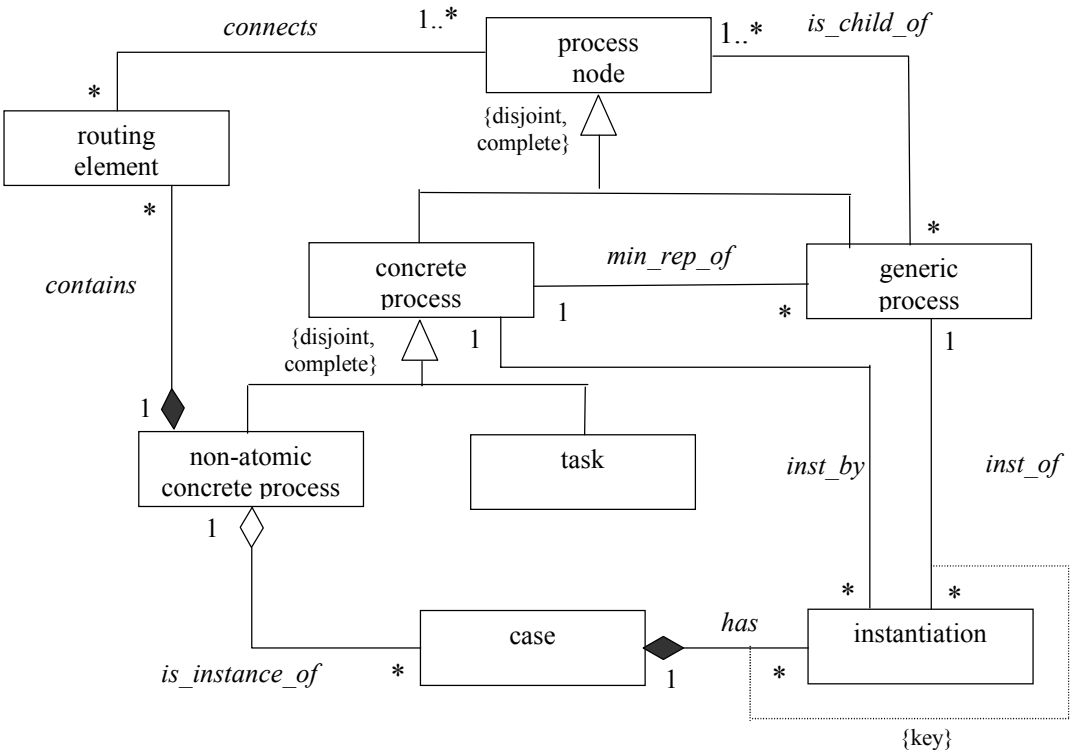

Figure 3: Class diagram describing the relationships between the main concepts used in this paper.

Figure 3 shows a class diagram, using the UML notation, relating the essential concepts used in this paper. The diagram shows that non-atomic concrete processes and tasks are specializations of concrete processes, i.e., both the class *non-atomic concrete process* and the class *task* are subclasses of the class *concrete process.* The two subclasses are mutually disjoint and complete. The class *process node* is a generalization of the class *concrete process* and the class *generic process*. The association *is_child_of* relates process nodes and generic processes. If the association relates a process node X and generic process Y, then X belongs to the process family of Y. Since process nodes can be in the process family of generic processes and a generic process can have many children (but at least one), the cardinality constraints are as indicated in the class diagram. A generic process has at least one child because it has a so-called *minimal representative* as indicated by the association *min_rep_of*. The minimal representative of a generic process is a concrete process which captures the essential characteristics of a process family. The minimal representative is needed to enable dynamic change and to generate aggregate management information. The class *routing element* links process nodes to non-atomic concrete processes. A non-atomic concrete process consists of process nodes (i.e., tasks, non-atomic concrete processes, and generic processes) which can be executed in a predefined way. Typical routing elements are the AND-split, AND-join, OR-split, and OR-join [51]. These elements can be used to enable sequential, parallel, conditional, alternative, and iterative routing. In the class diagram, we did not refine the class *routing element* because the approach presented in this paper is independent of the process modeling technique used. The association *contains* specifies the relation between routing elements and non-atomic concrete processes. Note that a routing element is contained

in precisely one non-atomic concrete process. The association *connects* specifies which process nodes are connected by each routing element. Note that the associations *contains* and *connects* can be used to derive in which non-atomic concrete processes a process node is used. The class *case* refers to the objects that are handled at run-time using a non-atomic concrete process description. The association *is_instance_of* relates each case to precisely one non-atomic concrete process. It is not possible to execute non-atomic concrete processes containing process nodes which are generic. Before or during the handling of a case, generic processes need to be instantiated by concrete processes. The class *instantiation* is used to bind generic processes to concrete processes for specific cases. Every instantiation corresponds to one case, one generic process, and one concrete process. Note that per case it is not allowed to have multiple instantiations for the same generic process.

There are many constraints not represented in the class diagram. Constraints that are important for the remainder are:

1. The relation given by the association *is_child_of* is acyclic.
2. The relation derived from the composition of association *contains* and association *connects* is acyclic.
3. The relation derived from the composition of the associations *contains*, *connects* and *is_child_of* is acyclic, e.g., a non-concrete process X is not allowed to contain a generic process Y if X is a child of Y.
4. The minimal representative of a generic process is also a child, i.e., the relation specified by the association *min_rep_of* is contained in the relation specified by *is_child_of*.
5. A generic process can only be instantiated by a concrete process if the concrete process is (indirectly) a child of the generic process.
6. For a case it is only possible to instantiate generic processes which are actually contained in the corresponding non-atomic concrete process.

The class diagram shown in Figure 3 contains three types of information:

1. *Routing information*
   The process description of each non-atomic concrete process. It specifies which tasks, non-atomic concrete processes, and generic processes are used and in what order they are executed. The classes *routing element*, *process node,* and *non-atomic concrete process* and the associations *contains* and *connects* are involved.
2. *Inheritance information*
   The relation between a generic process and its children. It specifies possible instantiations of generic processes by concrete processes, and concerns the classes *generic process*, *process node,* and *concrete process* and the associations *is_child_of* and *min_rep_of*.
3. *Dynamic information*
   Information about the execution of cases and instantiations of generic processes by concrete processes. It involves the classes *case* and *instantiation* and the associations *is_instance_of*, *has*, *inst_by*, and *inst_of*. Note that not the information itself dynamic but the information refers to the run-time dynamics of the system.

Today's workflow management systems do not support the definition of generic processes, i.e., it is only possible to specify concrete processes. In the remainder of

this section we focus on the modeling of generic processes using a combination of routing and inheritance diagrams.

3.2 Routing diagrams

A routing diagram specifies for a non-atomic concrete process the routing of cases along process nodes. Any workflow management system allows for the modeling of such diagrams. Examples of diagramming techniques are Petri-nets (COSA, INCOME, BaaN/DEM, Leu), Event-driven Process Chains (SAP/Workflow), Business Process Maps (ActionWorkflow), Staffware Procedures (Staffware), etc. Figure 4 shows the COSA Network Editor (CONE) while modeling the right-hand-side workflow process shown in Figure 1. COSA is a full-fledged workflow management system based on Petri nets. Figure 5 shows the same workflow process modeled with the workflow definer of Staffware. Staffware is one of the leading workflow management systems with more than 400.000 users. The two figures illustrate that although the diagramming techniques appear to be quite different, the essence is the same: the causal ordering of tasks using constructs such as choice, iteration, sequential composition, and parallel composition. Since there is no consensus on the diagramming technique to be used for workflow modeling, we use Petri nets to represent routing diagrams. Petri nets are a well-known technique for process modeling which combine formal semantics, expressive power, powerful analysis techniques, and an intuitive graphical representation [1,2,22].

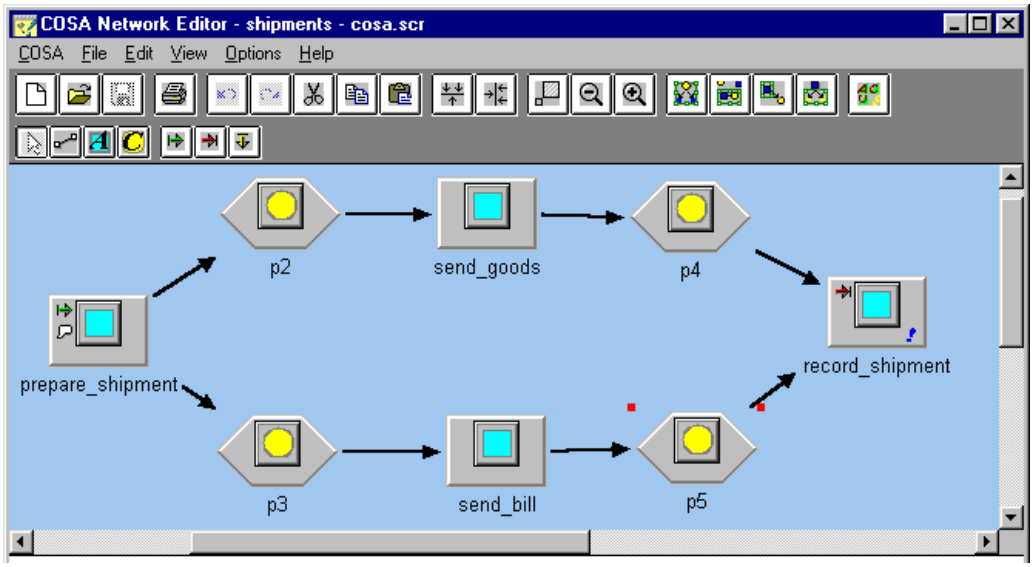

Figure 4: CONE: the design tool of COSA (COSA Solutions).

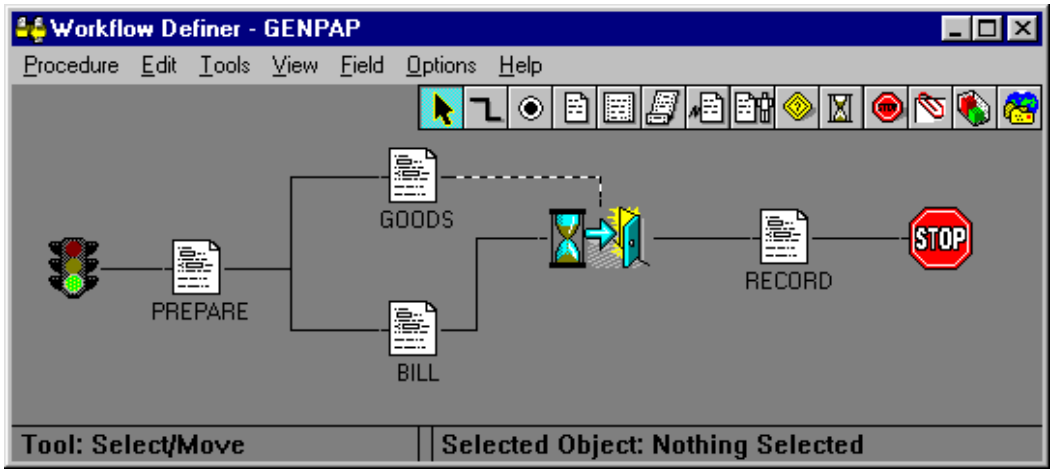

Figure 5: The workflow definer of Staffware (Staffware plc).

None of the diagramming techniques/workflow management systems discussed before supports generic processes. However, each of these diagramming techniques can be extended with generic processes. In this paper, we extend Petri-net-like routing diagrams [1,3,4,9,22] with generic processes.

A routing diagram specifies the contents of a *non-atomic concrete process* and consists of four types of elements:

1. *Tasks*
   A task is represented by a square and corresponds to a Petri-net transition.
2. *Non-atomic concrete processes*
   A non-atomic concrete process is represented by a double square and corresponds to a link to another Petri-net (i.e., a subnet).
3. *Generic processes*
   A generic process is represented by a square containing a diamond and corresponds to a link which can be instantiated by a process node.
4. *Routing elements*
   Routing elements are added to specify which process nodes need to be executed and in what order. Since we use Petri nets, routing elements correspond to places and transitions which are added for routing reasons only.

Figure 6 shows the four types of elements.

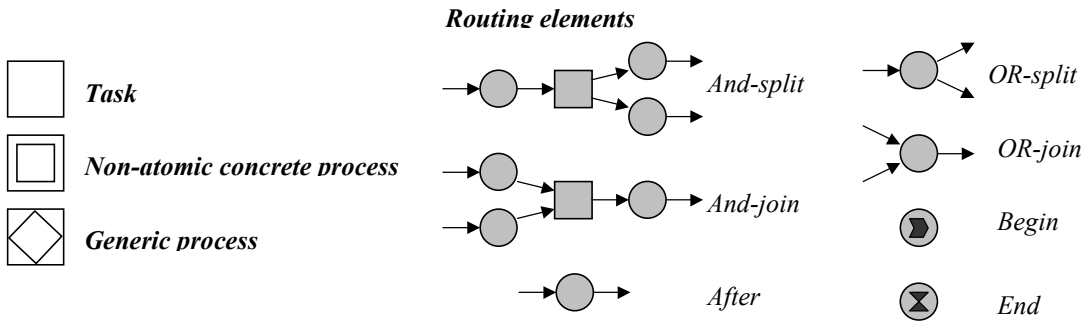

Figure 6: Symbols used in a routing diagram.

To illustrate the construction of routing diagrams we give some examples. Figure 7 shows the specification of the non-atomic concrete process *handle_insurance_claim*. This process consists of two tasks (*registration* and *pay_damage*), one non-atomic concrete process (*check_policy*), three generic processes (*check_damage*, *evaluate_claim*, and *reject_claim*), and several routing elements. Every insurance claim is first registered, then the policy and the damage are checked, followed by an evaluation which either results in a payment or in a rejection. Note that Figure 7 contains sequential, parallel, and conditional routing.

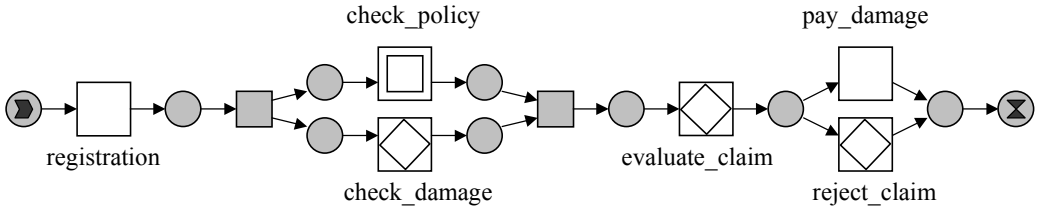

Figure 7: The non-atomic concrete process *handle_insurance_claim*.

The three generic processes shown in Figure 7 may correspond to several concrete processes. However, this information is not given in the routing diagram and will be specified in the corresponding three inheritance diagrams (see Section 3.3). Since *check_policy* is a concrete process which is not atomic, there is also a routing diagram specifying the contents of this process. Figure 8 shows the definition of *check_policy*. The process is completely sequential and contains only two tasks: *query_policy_data* and *query_payments_data*.

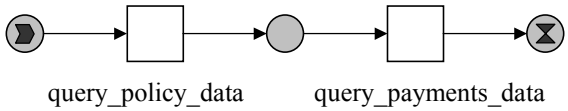

Figure 8: The sequential process *check_policy*.

Figure 9 shows a process with iteration. The definition of the non-atomic concrete process *phone* contains five tasks. First the task *dial* is executed followed by either the task *OK* (contact established) or *NOK* (no contact). If no contact was established there are two possibilities: either a letter is sent (*send_letter*) or task *dial* is executed again. Note that the only way to complete this process is sending a letter or successfully establishing contact via the phone.

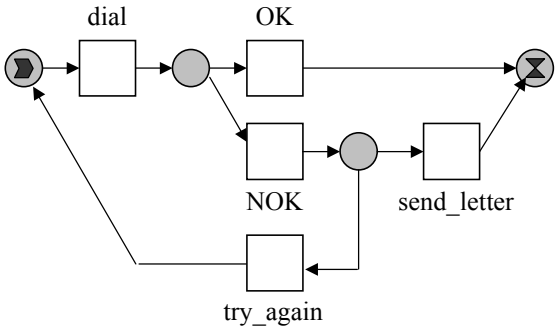

Figure 9: The process *phone*.

In the next subsection we will see that *phone* is a child of the generic process *reject_claim*. For completeness, we also define the non-atomic concrete process *check_car_damage_500+* which is indirectly a child of the generic process *check_damage*. Process *check_car_damage_500+* is a sequential process containing two tasks.

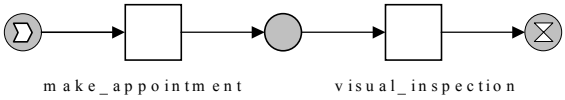

Figure 10: The process *check_car_damage_500+*.

The routing diagram shown in Figure 7 illustrates how a process definition language can be extended with generic processes. In principle, it is possible to use the diagramming technique offered by any workflow management system with a hierarchy concept and extend it with a new type of building block: the generic process. Note that the concepts presented in this paper do not rely on the Petri-net formalism. In fact, the results are independent of the process modeling technique. For simplicity we will assume that in a routing diagram there is one begin routing element, one end routing element, and that every process node (i.e., a task, a generic process, or a non-atomic concrete process) has exactly one input arc and one output arc. It is quite straightforward to extend the results to the situation without these assumptions.

3.3 Inheritance diagrams

In contrast to routing diagrams, today's products do not allow for inheritance diagrams to specify the process family corresponding to a generic process. The lack of such a concept in today's workflow management systems has many similarities with the absence of product variants in the early MRP/ERP-systems. These systems where based on the traditional Bill-Of-Material (BOM) and where burdened by the growing number of product types. Therefore, the BOM was extended with constructs allowing for the specification of variants [23,29,46,47]. Variants of a product type form a product family of similar but slightly different components or end-products. Consider for example a car of type X. Such a car may have 16 possible colors, 5 possible engines, and 10 options which are either present or not, thus yielding $16*5*2^{10}=81920$ variants. Instead of defining 81920 different BOM's, one generic BOM is defined.

Inspired by the various ways to define generic BOM's, we extend process models with inheritance diagrams allowing for the specification of process families.

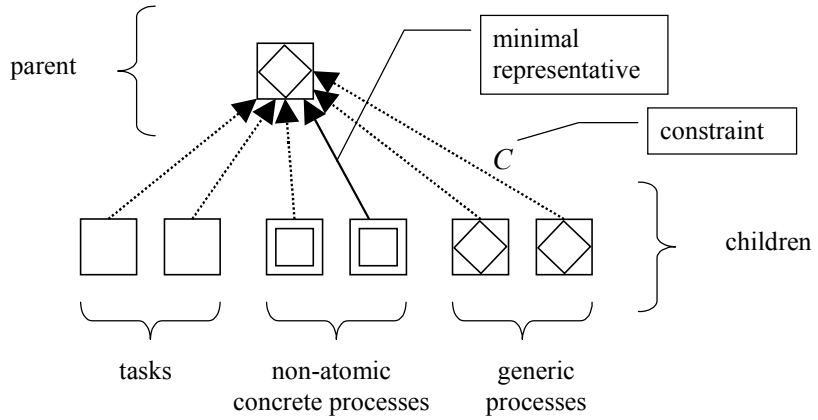

Figure 11: Symbols used in an inheritance diagram.

Figure 11 shows an inheritance diagram. The root of an inheritance diagram is a generic process called the *parent*. All other process nodes in the diagram are called the *children* and are connected to this parent. There are three types of children: tasks, non-atomic concrete processes, and generic processes. Each non-atomic concrete process in the inheritance diagram refers to a routing diagram describing the internal routing structure. Each generic child process in an inheritance diagram refers to another inheritance diagram specifying the process family which corresponds to this generic process. Note that the total number of inheritance diagrams equals the total number of generic processes. Every generic process has a child called the *minimal representative* of this task. This child is connected to the parent with a solid arrow. All the other arrows in an inheritance diagram are dashed. The minimal representative has all the attributes which are mandatory for the process family. One can think of this minimal representative as the default choice, as a simplified management version, or as some template object. The actual interpretation of the minimal representative depends on its use. The minimal representative can be considered to be the superclass in an object-oriented sense [6]. All other children in the inheritance diagram should be subclasses of this superclass. For execution, generic processes are instantiated by concrete processes using the relations specified in the inheritance diagram. However, in many cases it is not allowed to instantiate a parent by an arbitrary child. Therefore, it is possible to specify constraints as indicated in Figure 11. These constraints may depend on two types of parameters: (1) *case variables* and (2) *configuration parameters*. The case variables are attributes of the case which may change during the execution of the process (cf. [4]). Configuration parameters are used to specify that certain combinations of instantiations are not allowed. These parameters can be dealt with in a way very similar to the parameter concept in [46] for the generic BOM.

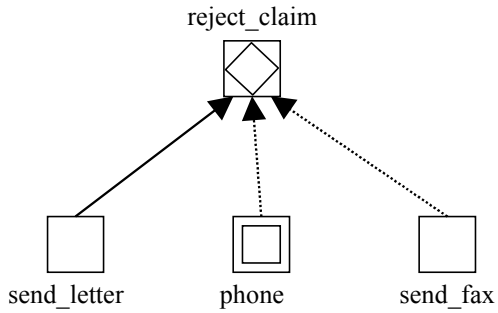

Figure 12: An inheritance diagram for the generic process *reject_claim*.

Figure 12 shows an inheritance diagram with parent *reject_claim* and three children (the tasks *send_letter* and *send_fax*, and the non-atomic concrete process *phone*). The task *send_letter* is the minimal representative of *reject_claim*. In this case all children are concrete. Note that the generic process *reject_claim* was used in the process *handle_insurance_claim* (Figure 7). As Figure 12 shows, this generic process can be instantiated by the tasks *send_letter* or *send_fax*, or the non-atomic concrete process defined in the routing diagram shown in Figure 9.

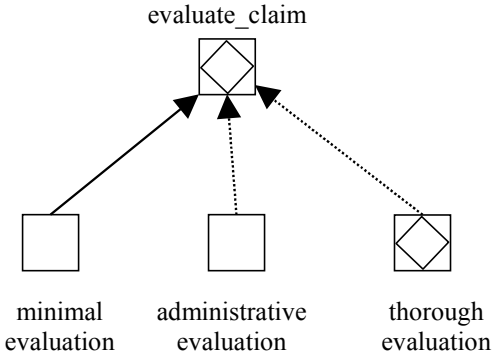

Figure 13: The process family corresponding to the generic process *evaluate_claim*.

The process *handle_insurance_claim* shown in Figure 7 also uses the generic process *evaluate_claim*. Figure 13 shows the inheritance diagram of *evaluate_claim*. Note that the generic process *thorough_evaluation* is a subclass of *evaluate_claim*.

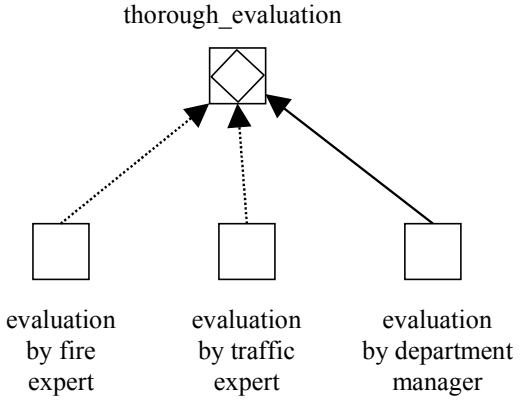

Figure 14: An inheritance diagram for the generic process *thorough_evaluation*.

Figure 14 shows that there are three ways to evaluate a claim thoroughly. Note that the process family of *evaluate_claim* consists of five children. Generic process *evaluate_claim* in Figure 7 is instantiated by one of these children, i.e., one of the two tasks in Figure 13 or one of the three tasks in Figure 14.

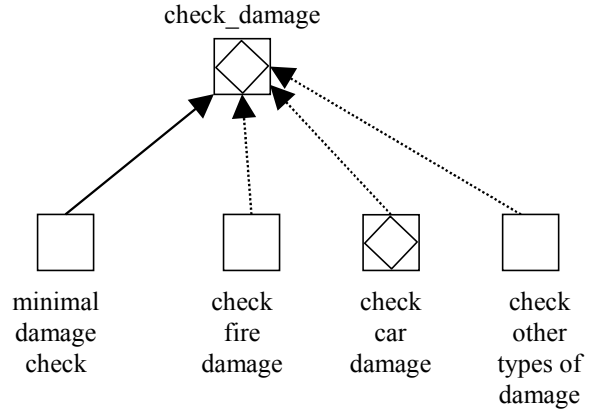

Figure 15: An inheritance diagram for the generic process *check_damage*.

Generic process *check_damage* is executed in parallel with the non-atomic concrete process *check_policy* (see Figure 7). Figure 15 shows the inheritance diagram of *check_damage*. The subclass *check_car_damage* is defined in Figure 16. There are five ways to check the damage. In case of car damage, there are two possibilities: the task *check_car_damage_500-* or the non-atomic concrete process *check_car_damage_500+* defined in Figure 10 is executed.

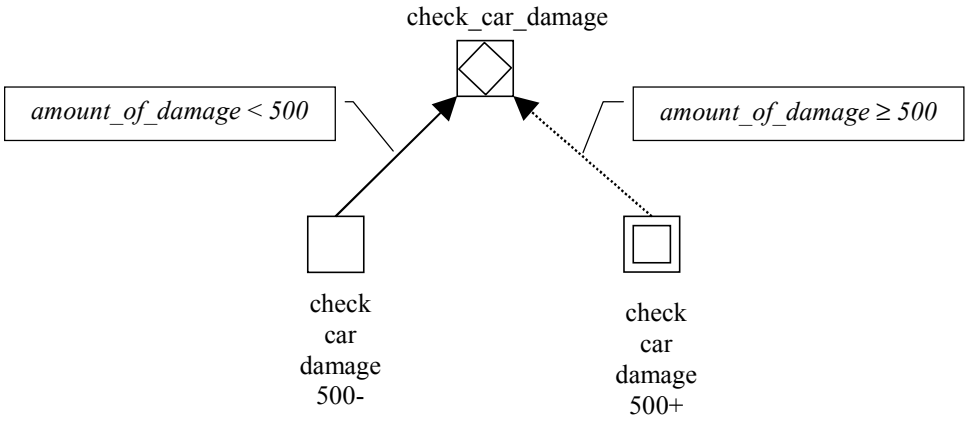

Figure 16: An inheritance diagram for the generic process *check_car_damage*.

Inheritance diagrams specify, for each generic process, possible candidates for instantiation. However, in many cases it is not allowed to instantiate a parent by an arbitrary child. Therefore, it is possible to specify constraints as indicated before. These constraints may depend on two types of parameters: (1) *case variables* and (2) *configuration parameters*. The case variables are attributes of the case which may change during the execution of the process. For example, the choice between the two children in Figure 16 clearly depends on the case variable. Therefore, we added the

constraint *amount_of_damage<500* in Figure 16. This constraint specifies that generic process *check_car_damage* may only be instantiated by the non-atomic concrete process *check_car_damage_500*- if the case variable *amount_of_damage* indicates that the damage is smaller than 500 dollar. Configuration parameters are used to specify that certain combinations of instantiations are not allowed. For example, if *evaluate_claim* in Figure 7 is instantiated by *administrative_evaluation*, then *reject_claim* should be either instantiated by *send_letter* or *send_fax*. Figure 17 shows how one can use the configuration parameter *config_par_ae* to avoid that *reject_claim* is instantiated by *phone* if *evaluate_claim* is instantiated by *administrative_evaluation.* In this paper, we propose a very simple language for specifying constraints with respect to the instantiation of generic processes: Every arc in an inheritance diagram may be augmented with a Boolean condition (the default condition is true) specified by a logical expression using case variables and configuration parameters. The case variables for a specific case (i.e., process instance) are set and updated by the actual workflow process. The configuration parameters are also case-specific. One can think of these variables as free variables that are set while instantiating generic processes, i.e., they are outside of the scope of the actual workflow process. The use of configuration parameters can be quite complex. Fortunately, we can use the concepts defined for the generic BOM. See [46,47] for more information on the use of configuration parameters. In addition to constraints it is possible to give suggestions for instantiation. Often there is a trade-off between several allowed alternatives. Therefore, it is useful to extend inheritance diagrams with information on the effect of specific instantiations on key performance indicators (e.g. time, costs, quality of service, and flexibility).

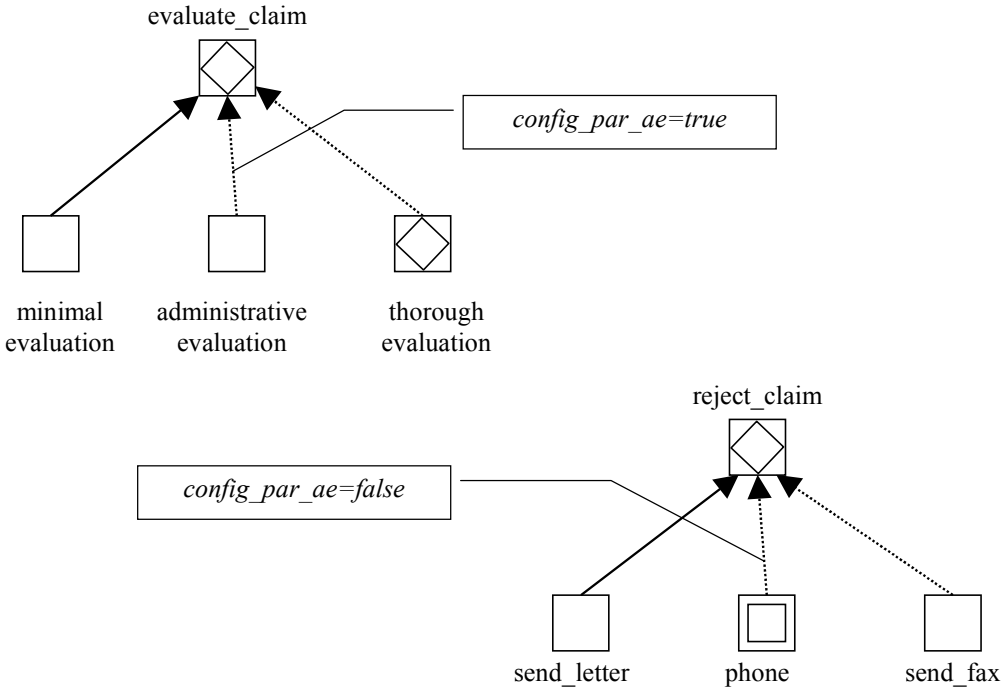

Figure 17: The process families *evaluate_claim* and *reject_claim* extended with configuration parameter *config_par_ae*.

Note that, in principle, the construct shown in Figure 16 can be realized using a an OR-split in traditional routing diagram: simply add an OR-split based on the case

variable *amount_of_damage* which enables the appropriate task/subprocess. Even the construct shown in Figure 17 can be handled in traditional routing diagram by adding a new case variable with a role similar to configuration parameter *config_par_ae.* Although it is possible to realize this flexibility without generic processes and inheritance diagrams, there are some essential differences. First of all, the use of inheritance diagrams reduces the complexity. Instead of squeezing all variants into a single very complex routing diagram, the variations are handled in a second dimension tailored towards handling many variants. Second, by using inheritance diagrams there is no need to change the routing diagram every time a new variant pops up. If the intrinsic structure is not changed, it suffices to simply add a new child to the corresponding process family. Third, the inheritance diagrams can be used in multiple routing diagrams, i.e., the inheritance diagrams are truly orthogonal to the routing diagrams thus enabling reuse. Note that this imposes some restrictions on the use of case variables and configuration parameters in the conditions specified in the inheritance diagrams. For example, by using a case variable *amount_of_damage* in Figure 16 it is required that every routing diagram which uses the generic process *check_car_damage* sets the variable to the appropriate value. Note that these restrictions are quite reasonable because the diagrams are used within a given organizational context with common concepts. Otherwise, reuse is difficult to achieve anyway. Moreover, one could also use the principle that a condition evaluates to true if one of the case variables is not set. This way the conditions only apply to the relevant processes. A fourth difference between using inheritance diagrams and traditional routing is the improved ability to support dynamic change and supply succinct management information: The inheritance diagrams help to localize change and abstract from individual variations, and the concept of the minimal representative allows for the correct and automatic migration of instances.

3.4 Navigation

Workflow processes encountered in practice typically contain dozens of tasks. If the workflow management also allows for the modeling of variants using the notions described in this paper, it will become difficult not to ‘get lost’. Therefore, the workflow management system should support navigation tools to find and keep track of workflow descriptions. Basically, there are two ways to browse through the workflow processes. First of all, it is possible to use the associations *contains* and *connects* (see the class diagram in Figure 3). These associations show where process nodes are used. Secondly, it is possible to use the association *is_child_of* to see inheritance relationships. The two navigation dimensions are inspired by the work reported in [40].

The *is-part-of* and *contains* arrows correspond to the navigation dimension based on the routing diagrams. The *generalizes* and *specializes* arrows correspond to the navigation dimension based on the inheritance diagrams. The combination of both navigation dimensions is particularly powerful. Users of the workflow management system can browse through the workflow process definitions at various levels of abstraction, which greatly enhances the ability to combine existing process descriptions and to build new variants which can be used by others. Figure 18 shows an overview of all process nodes for the example presented in this paper. Each of the two navigation dimensions is shown with a different type of arrow.

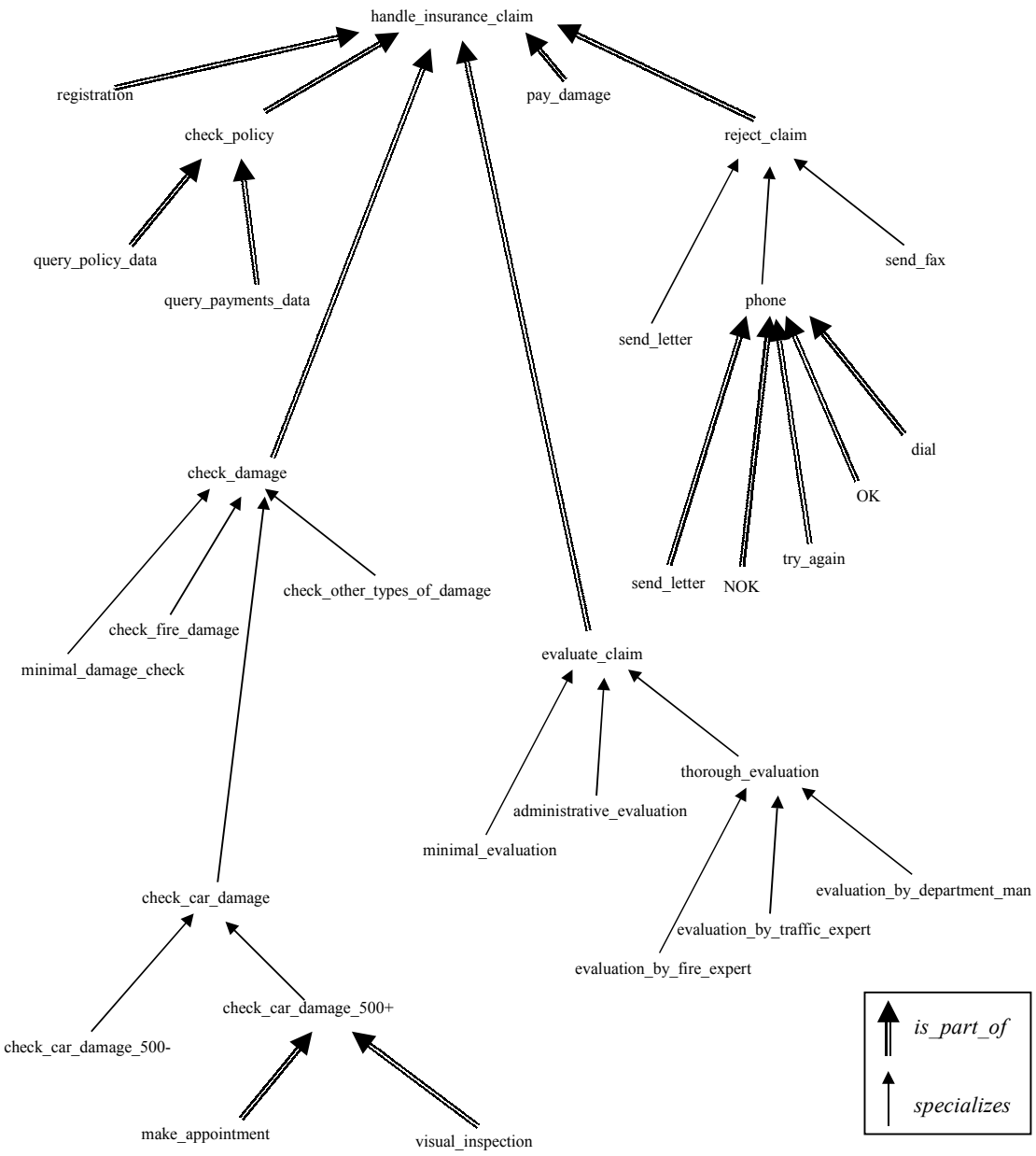

Figure 18: All process nodes with links in two dimensions.

The graph shown in Figure 18 has a tree-like structure. In general this is not the case. A process node can appear in many routing and inheritance diagrams. In fact, the use of generic processes will stimulate reuse and typically results in many non-tree-like interconnections. Note that graphs such as the one shown in Figure 18 have to be acyclic. This is a direct consequence of the constraints mentioned in Section 3.1.

4 Execution and instantiation

To execute a case according to a non-atomic concrete process which contains generic processes, the generic processes have to be instantiated. The moment of instantiation can be at entry-time or at run-time depending on the kind of changes allowed. Consider for example a case which needs to be executed according to the routing

diagram shown in Figure 7. To handle this case the three generic processes need to be instantiated by concrete ones. Assume that at entry-time *check_damage* is instantiated by *check_car_damage_500+* and *evaluate_claim* is instantiated by *evaluation_by_traffic_expert*. This results in the situation shown in Figure 19.

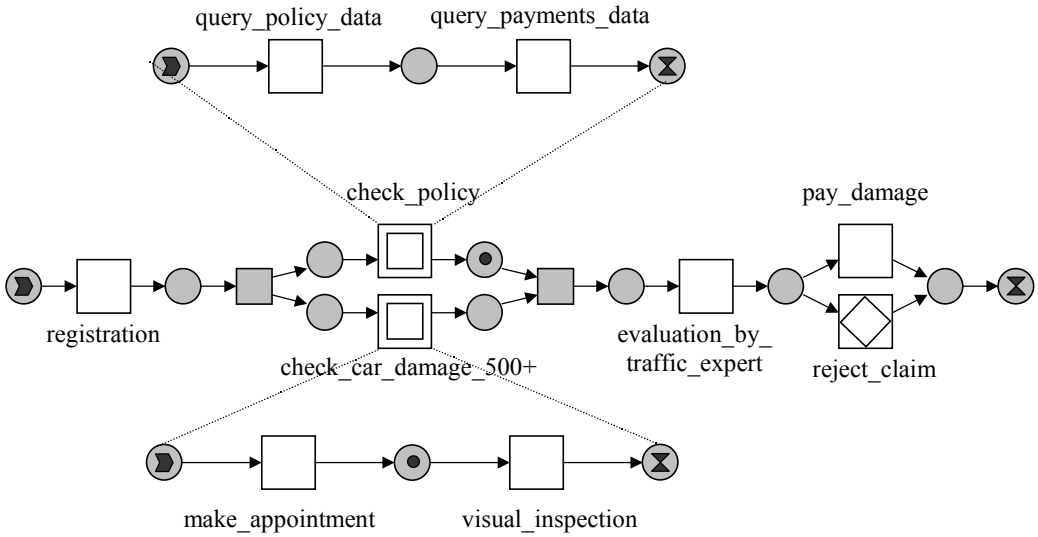

Figure 19: Two of the three generic processes have been instantiated.

In Figure 19, the two tokens show that the policy has been checked and that the insurance claim is waiting for visual inspection. Generic process *reject_claim* is not instantiated yet. The moment of instantiation can be postponed until the tasks *visual_inspection* and *evaluation_by_traffic_expert* have been executed. Figure 20 shows the situation after instantiation of *reject_claim* by *phone*. Note that the case is in the state directly following an unsuccessful phone call.

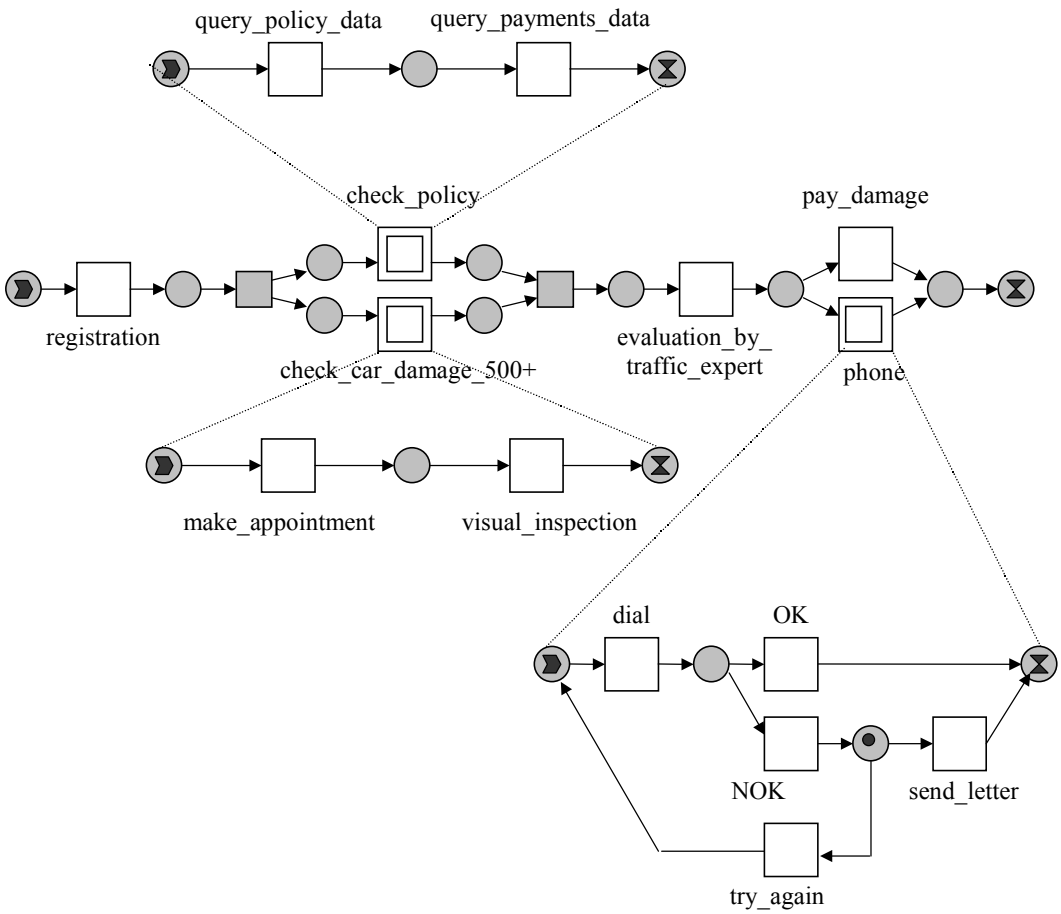

Figure 20: The generic process *reject_claim* has been instantiated by the non-concrete process *phone*.

In Section 5, a classification of change was given. Based on this classification we will describe the enactment of generic workflow process models. First, we consider the situation of ad-hoc change. For ad-hoc change there are two possibilities: at entry time the process is fixed or on-the-fly changes are possible. If changes are only allowed at entry time, all generic processes have to be instantiated by concrete processes before execution starts. If on-the-fly changes are allowed, instantiation may be postponed. Note that ad-hoc change may lead to two possible situations: (1) the required alternative already exists and can be instantiated directly, (2) the desired change is unique in the sense that it is not encoded in one of the inheritance diagrams. In the latter case the *change region* can be a generic process, a task, a non-atomic concrete process, or a selected part of a routing diagram (i.e., non-atomic concrete process). If the change region is a generic process, it is quit straightforward to handle the change. By adding a new alternative in the corresponding inheritance diagram, the change becomes possible. In all other cases, a new generic process has to be introduced. Consider for example the concrete process *check_policy* in Figure 7. If this process has to be replaced by a new process *check_policy_special*, then the concrete process *check_policy* is replaced by a new generic process *check_policy_generic* and an inheritance diagram containing *check_policy_generic* as a parent and *check_policy* and *check_policy_special* as children is added (see Figure 21). Compared to the situation where each case has its own private process, this offers several advantages. First of all, there is no redundancy. There is one generic process model and per case

only the instantiations are stored. Secondly, it is possible to restrict possible changes, e.g., routing diagrams can be extended with information about which parts may be subject to ad-hoc changes.

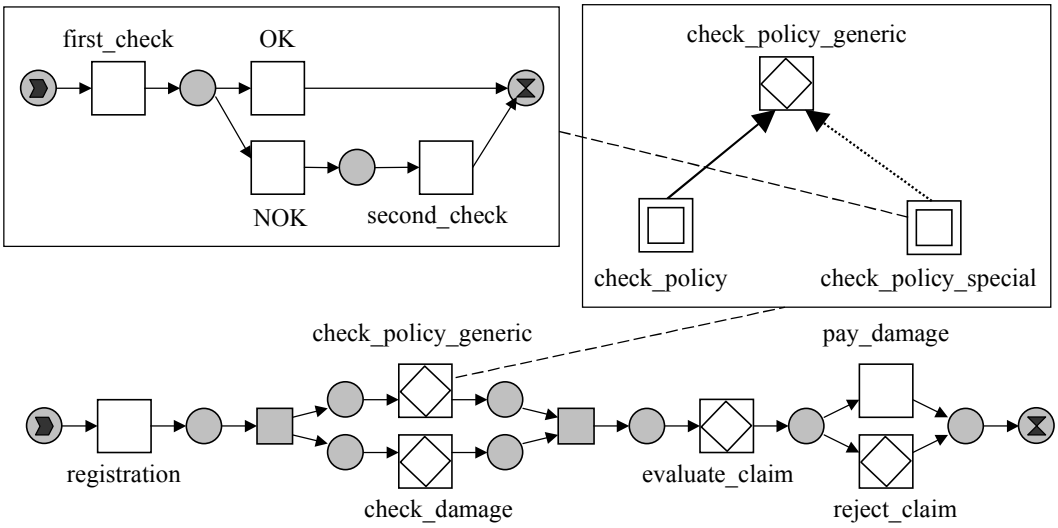

Figure 21: The effect of an ad-hoc change on the non-atomic concrete process *handle_insurance_claim*.

Generic process models can also be used to support evolutionary changes. The only difference with ad-hoc change is that all new cases are instantiated in the same way. Compared to the situation where there is just a versioning mechanism, less data needs to be stored (no redundancy, instantiations can be indexed) and it is easy to keep track of changes. For example, if a task is replaced by another task in the definition of a non-atomic concrete process (i.e., a routing diagram) with 100 tasks, it is not necessary to define a complete new version of the process which is for 99% similar. Just replace the task by a generic process and create an inheritance diagram containing the two tasks as children. If the change also effects existing cases, a restart or a transfer is needed (see **Error! Reference source not found.**).

5 Dynamic change

Both for ad-hoc and evolutionary change the generic workflow process model, as defined in this paper, can be applied. *The workflow model shown in Figure 7 has 75 possible variants!* If all variants had been defined separately, the whole would be much more difficult to manage. Many workflow processes have thousands of variants. Today's workflow management systems are not capable of dealing with such processes, thus forcing the users to handle the changes outside of the system or limiting the application of these systems to very specific processes. The generic workflow process model presented in this paper provides a partial solution for these problems.

The problem of dynamic change was introduced using Figure 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 *p2* and *p5* (right-hand side of Figure 1) cannot be mapped onto a state in the sequential process (left-hand side). Putting a token in *s1*, *s2,* or *s3* will result in the double execution of task *C*. Putting a token in *s3*, *s4,* or *s5* will result in the skipping of (at least) task B. The problem identified does not only apply to the situation where the degree of parallelism is changed. For example swapping tasks or removing parts 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 mortgages and insurances 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.

There are many similarities between dynamic change 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. [16]) 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, it is not always possible to transfer. 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.

The dynamic change problem was first described by Ellis, Keddara and Rozenberg in 1995 [20]. In the same paper, a technique based on so-called "change regions", is proposed to avoid the anomaly illustrated by Figure 1. There has been some follow up work addressing this problem [13,21,34]. In addition there are several papers on workflow change, not addressing and/or avoiding the problem by a fixed set of transformation rules [17,27,30,36,41,49,50,52].

Independent of the approach used, the following two issues constitute a policy for dynamic change:

- When to jump from the old process to the new process definition?
- Which state to jump to?

A good policy for the example shown in Figure 1 is the following. The right-hand process will jump in every state except the state with a token in p2 and p5. State *p1* is mapped onto *s1*, *p2+p3* onto *s2*, *p3+p4* onto *s3*, *p4+p5* onto *s4*, and *p6* onto *s5*. (Note that a shorthand notation is used to denote states.)

In a generic process model, the dynamic change problem boils down to migrating instances between different members of the same process family. Note that the concept of generic processes helps to limit the scope of a change. In a way it is a predefined "change region" [20]. Recall that if a part is changed which does not correspond to a generic process, then a generic process is introduced. (See for example the change illustrated by Figure 21.) The essence of a change always refers to transferring (parts of) cases between children of a generic process. Although the concept of generic processes gives a handle to tackle the problem, it does not really solve it. In fact, if a process family has many members, say *n*, there are *n(n-1)* potential transfers. To limit the problem, we propose to exploit the role of the minimal representative. Any transfer between two members of the same process family is executed via the minimal representative, i.e., first the instance is mapped from the old child onto the minimal representative, and then it is mapped onto the new child. Note that this results in *2(n-1)* possible transfers. If a new variant is added, only the transformation from the new variant to the minimal representative and vice versa need to be added and no knowledge of the other variants is needed. A solution with direct jumps would require knowledge of all other variants. One might argue that only a few of the potential transfers are relevant. However, to truly support reusability all possible transfers should be defined. Clearly there are also drawbacks associated with the indirect transfer via the minimal representative. First of all, if the minimal representative contains little information, a lot of knowledge is lost during the transfer. It is clear that a transfer between two children with a state space of thousands of states via a minimal representative with only a dozen states is not likely to be a success (because of the loss of information). Secondly, additional problems are introduced the moment a new minimal representative is introduced. Therefore, it is vital to carefully define the minimal representative.

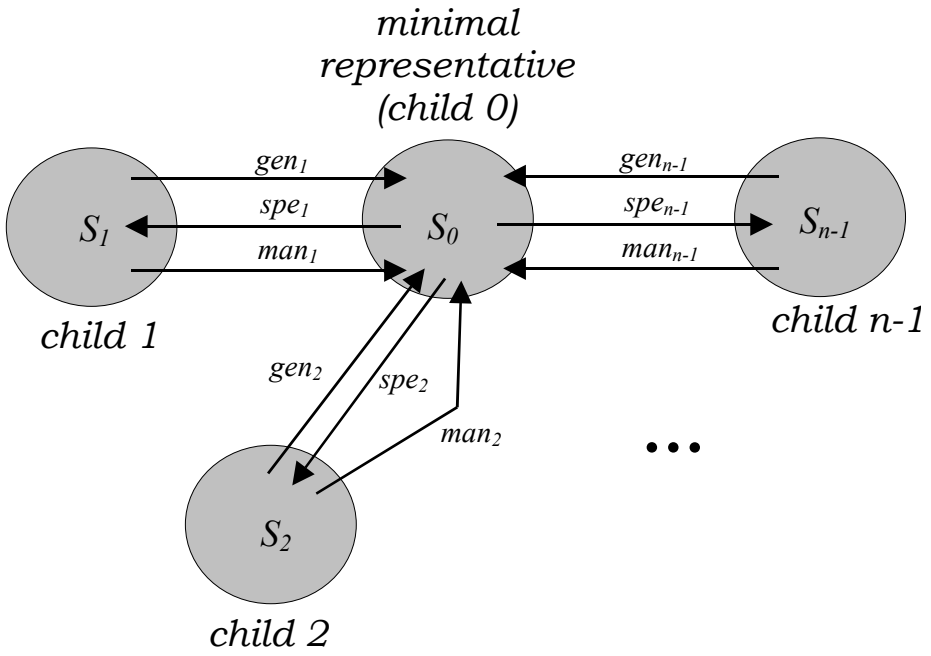

Figure 22: The relation between the state spaces of the minimal representative and its fellow children.

Figure 22 illustrates the use of the minimal representative. Each child (including the minimal representative) has a state space. S_0 is the state space of the minimal

representative (child 0). Child i has state space S_i. The partial function $gen_i \in S_i \nrightarrow S_0$ maps selected states of child i onto the minimal representative. The function is partial because from some states it is desirable to postpone the jump, i.e., state space S_i is partitioned into $S_i^J = dom(gen_i)$, the set of "jump states", and $S_i^W = S_i \setminus S_i^J$, the set of "wait states". There is a similar function to map states of the minimal representative onto the states of a specific child: $spe_i \in S_0 \nrightarrow S_i$. This function is also partial and partitions the states of S_0 into jump states ($S_0^{J,i} = dom(spe_i)$) and wait states ($S_0^{W,i} = S_0 \setminus S_i^{J,i}$) relative to child i. A transfer from one child (i) to another child (j) typically involves a generalization step (i.e., gen_i) and a specialization step (i.e., spe_j). The functions of type $man_i \in S_i \rightarrow S_0$ shown in Figure 22 will be used to generate management information and should be ignored for the moment.

Suppose a case needs to be transferred from child i to child j and the state of the case is $s \in S_i$. If $s \in S_i^W$, no transfer is possible. If $s \in S_i^J$ and $gen_i(s) \in S_0^{J,j}$, then there is no reason to postpone the jump to the new process. The new state in the process corresponding to child j is $spe_j(gen_i(s))$. If $s \in S_i^J$ and $gen_i(s) \in S_0^{W,j}$, there are two policies possible: (1) the transfer is postponed (non-eager), or (2) the case is migrated to the minimal representative and is transferred the moment it reaches a state in $S_0^{J,j}$ (eager). If the change affects several parts of the workflow process definition and multiple generic processes are involved, there is a similar choice. Either the transfer is postponed until all parts are ready (non-eager) or the transfer is executed on a part-by-part basis (eager). At this moment, the policy to execute the transfer on a part-by-part basis but postponing parts which cannot go directly to the new corresponding child seems to be the most attractive policy. However, more empirical data is needed to substantiate this.

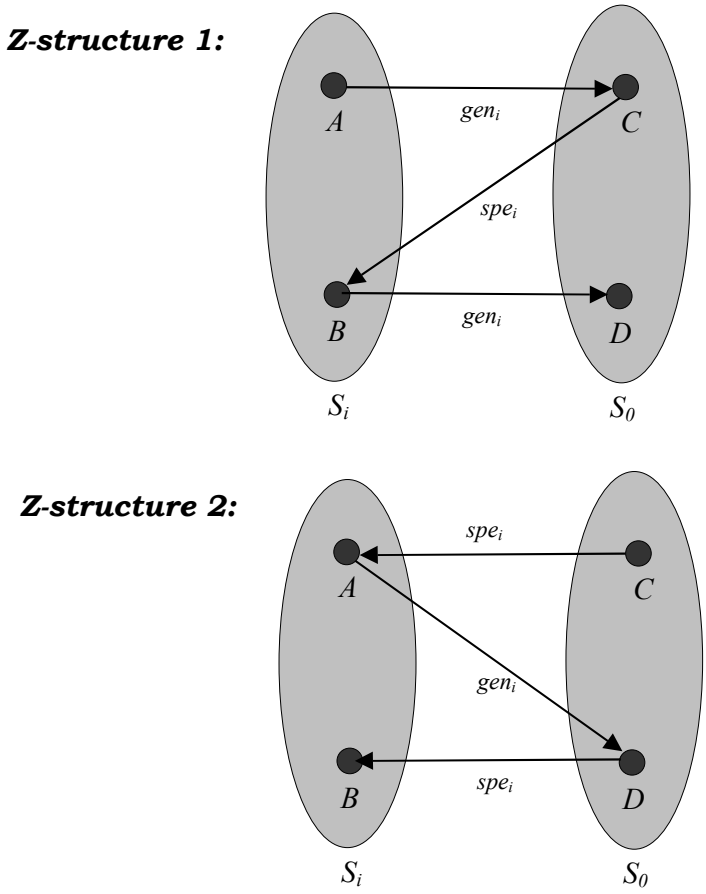

Figure 23: Two Z-structures that are not allowed.

Not every set of generalization (gen_i) and specialization (spe_i) functions is allowed. Constructs which have a so-called "Z-structure" are not allowed. A "Z-structure" is the situation where two distinct states are mapped onto two other distinct states in one direction (e.g. generalization) but in the reverse direction (e.g. specialization) one of the states is mapped onto the other one. Figure 23 shows the two possible Z-structures. In the first Z-structure there are two states *A* and *B* which are mapped onto respectively *C* and *D* by the generalization function (gen_i). However, the specialization function (spe_i) maps *C* onto *B* instead of *A*. This structure is not allowed because by simply moving a case up (2x) and down, the state in *both* processes has changed (in the left-hand process it moved from *A* to *B* and in the right-hand process it moved from *C* to *D*). Note that it is not possible to strengthen the requirement and demand that for any state *s*: $spe_i(gen_i(s)) = s$, because multiple states in the child process *i* can be mapped onto one state in the minimal representative. In the second Z-structure shown in Figure 23, the roles of the generalization (gen_i) and specialization (spe_i) functions have been swapped and similar arguments apply. The absence of these Z-structures is the minimal requirement any dynamic change should satisfy. There are generally additional requirements that need to be satisfied. Suppose that the right-hand-side process in Figure 1 is the minimal representative and the left-hand-side process is the child 1. Assume that gen_1 is defined as follows: *s1* is mapped onto *p6*, *s2* is mapped onto *p1*, *s3* is mapped onto *p6*, *s4* is mapped onto *p1*, and *s5* is mapped onto *p6*. Moreover, spe_1 is defined as follows: *p1* is mapped onto *s2*, and *p6* is mapped onto *s1* (the other states are wait states). Clearly, this does not make any sense. Nevertheless, it does not contain any Z-structures. Stronger notions are context

dependent and are difficult to define for any process modeling technique. (Recall that the concepts in this paper are modeling technique independent.) Therefore, we refrain from more advanced constraints that should be satisfied by the set of generalization (gen_i) and specialization (spe_i) functions.

In Section 2, we identified three ways to deal with existing cases: (a) restart, (b) proceed, and (c) transfer. Thus far, we primarily discussed the problems resulting from the latter policy (i.e., dynamic change). However, the approach presented in this section also works for the other two policies. For the restart policy (a), all states of the old process *i* are mapped onto the initial state of the minimal representative (i.e., $S_I^J = S_i$ and for all $s \in S_i$: $gen_i(s) = s_{init}$ where s_{init} is the initial state) and the initial state of the minimal representative is mapped onto the initial state of the new process *j* (i.e., $spe_j(s_{init}) = s'_{init}$ where s'_{init} is the initial state of child *j*). For the proceed policy (b), all states are wait states, i.e., $S_i^J = \emptyset$. Clearly, the approach presented is quite general and can be extended in many ways. For example, it is possible to deal with hierarchical structures in an efficient way since change is limited to the generic parts of the process. It is also possible to allow for changes of the minimal representative. Simply add a generalization function from the old minimal representative to the new one and a specialization function from the new minimal representative to the old one. By taking the appropriate function compositions, it is possible to remove or skip the old minimal representative.

6 Management information

Changes typically lead to multiple variants of the same process. For evolutionary change the number of variants is limited. In fact, if all cases are transferred directly after a change, there is just one active variant. However, if the proceed policy is used or transfers are delayed, there are multiple active variants. If the average flow time of cases is long and changes occur frequent, there can be dozens of variants. Ad-hoc change may result in even more variants. 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. To manage a workflow process with different 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 the manager with tools to obtain a condensed but accurate view of the workflow processes. In Figure 2, it was pointed out that we need some kind of ‘Greatest Common Denominator’ (GCD) or ‘Least Common Multiple’ (LCM) for the children in a product family. At the moment, only intuitive notions exist for the GCD and LCM. However, we can use the same approach as we used to tackle the dynamic change problem and use the minimal representative as the aggregated view.

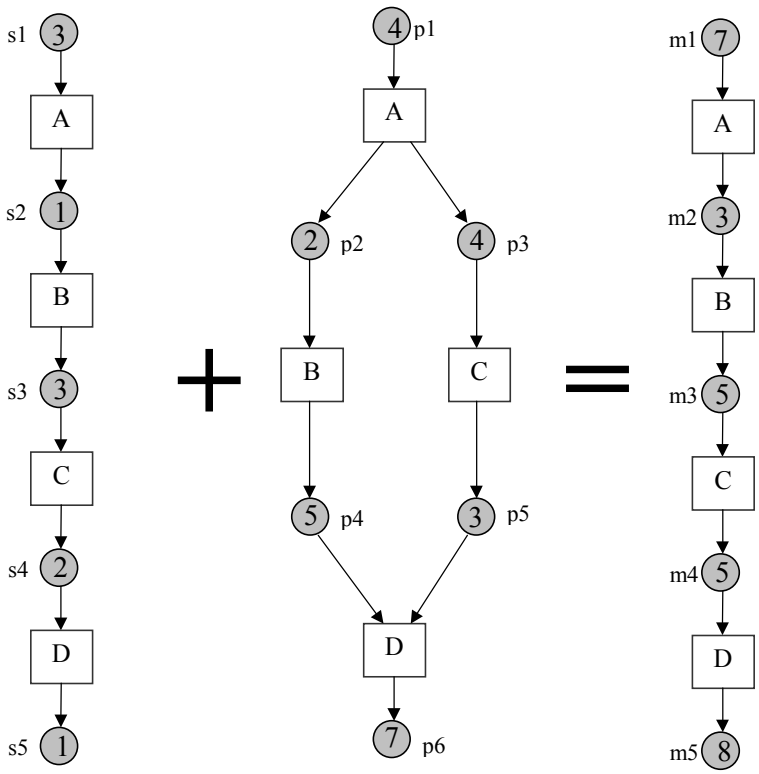

Figure 24: Aggregated management information mapped onto a sequential minimal representative.

To use the minimal representative as the aggregated view, we need to map all states from all children of the process family onto the state space of the minimal representative. The generalization functions (gen_i) provide such a mapping for the jump states but not for the wait states. Therefore, we introduce a new function for each child *i* (except the minimal representative): man_i. The functions of type $man_i \in S_i \rightarrow S_0$ are total and should satisfy the following requirement: for all $s \in S_i^J$ we have $man_i(s) = gen_i(s)$. I.e., the mapping used for dynamic change and the mapping used for management purposes should agree on the jump states. Again, the solution is surprisingly simple. However, the applicability heavily depends on the quality of the minimal representative and the functions of type man_i. Figure 24 shows an alternative to the approach used in Figure 2. In this case, the sequential process is taken as the minimal representative. The mapping of tokens from the left-hand-side process is clear (the state with a token in *s1* is mapped onto the state with a token in *m1*, etc.). In fact, the left-hand-side process and the right-hand-side process are identical and the places are named different for presentation reasons only. Mapping states from the process in the middle is more involved. For the jump states the following mapping seems to be reasonable: *p1* is mapped onto *m1*, *p2+p3* is mapped onto *m2*, *p3+p4* is mapped onto *m3*, *p4+p5* is mapped onto *m4*, and *p6* is mapped onto *m5*. In the previous section, state *p2+p5* was classified as a wait state because there is no intuitively corresponding state in the sequential process. Mapping *p2+p5* onto *m2* will lead to management information which is too pessimistic: *C* is already executed but this information is lost in the aggregated view. Mapping *p2+p5* onto *m4* will lead to management information which is too optimistic: *B* is not executed yet but this information is lost in the aggregated view. Mapping *p2+p5* onto *m3* combines the disadvantages of the previous two choices: it indicates that B has been executed and C

not, while in reality it is the other way around. This example shows that quality of the management information heavily depends on the minimal representative. The numbers indicated in Figure 24 are based on the assumption that cases are executed in a first-in-first-out order. This assumption combined with the numbers indicated for the parallel process implies that there are no cases in the state *p2+p5*. In this particular situation, the aggregated view does not depend upon the choice with respect to *p2+p5*. In general, an unfortunately chosen minimal representative will lead to misleading management information.

7 An approach based on inheritance

The approach presented in the previous sections is very general and makes no assumptions about the modeling language and the workflow management system to be used. To make the approach more concrete we show how to come up with the appropriate generalization (gen_i), specialization (spe_i), and management-information (man_i) functions. For this purpose we propose to use the *inheritance preserving transformation rules* presented in [6,14].

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 [14] 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 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. To our knowledge, the work presented in [6,14] is the only work which 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* (see Appendix). 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. Consider for example the work by Malone et al. [40] where inheritance is defined for tasks and processes. Malone et al. [40] also provide tool support for navigating through a space of processes using specialization and generalization links (see also Section 3.4). 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 (InConcert

[32]) 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 [6,14] and illustrate the usefulness by tackling the problems related to change. The inheritance notions can be used to construct a minimal representative and the appropriate generalization (gen_i), specialization (spe_i) and management-information (man_i) functions.

First we define four inheritance notions for workflow processes (i.e., processes defined by routing diagrams). 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. For distinguishing x and y we only consider the old tasks (i.e., the tasks already present in y). All other tasks are renamed to τ. One can think of these tasks as silent, internal, or not observable. Since *branching bisimulation* [24] 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*. 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 blocks all new tasks and then compares the observable behavior. This mechanism leads to the following notion of inheritance.

> *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.*

Intuitively, this definition conforms to *blocking* or *encapsulating* 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 consider only the old ones.

> *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.*

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

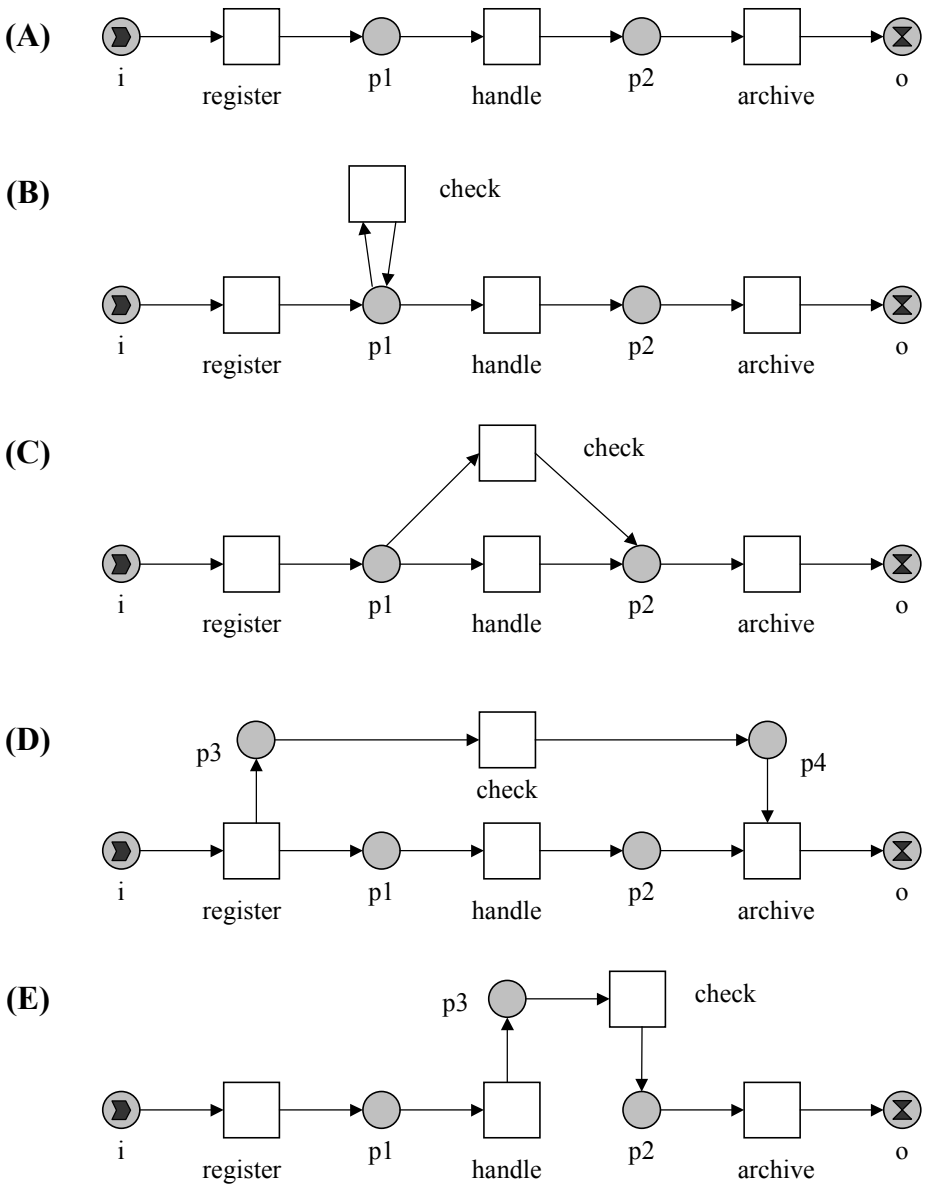

Figure 25: Five routing diagrams describing variants of a simple workflow process.

Figure 25 shows the routing diagrams of five similar workflow processes. We will use these routing diagrams to explain the difference between protocol inheritance and projection inheritance. 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 protocol inheritance; if task *check* is blocked, then the two processes behave equivalently (i.e., are branching bisimilar [14,24]). Workflow process (B) is also a subclass of workflow process (A) with respect to projection inheritance; if every execution of task *check* is abstracted from, then the observable behaviors are equivalent. Workflow process (C) is a subclass of workflow process (A) with respect to protocol inheritance but not a subclass with respect to projection inheritance; blocking task *check* results in two equivalent processes but hiding task *check* introduces the possibility to skip task

handle and thus change the actual behavior. Workflow process (D) is a subclass of workflow process (A) with respect to projection inheritance but not a subclass with respect to protocol inheritance; blocking task *check* introduces a deadlock, but hiding this task results in two equivalent processes. Workflow process (E) is a subclass of workflow process (A) with respect to projection inheritance but not a subclass with respect to protocol inheritance; the detour via task *check* can be hidden but not blocked without changing the observable behavior.

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 is a subclass of another workflow process under *protocol/projection inheritance* if by both hiding and blocking one cannot detect any differences, i.e., it is a subclass under both protocol and projection inheritance. In Figure 25 workflow process (B) is a subclass of workflow process (A) with respect to protocol/projection inheritance. The two mechanisms can also be used to obtain a weaker form of inheritance. A workflow process is a subclass of another workflow process under *life-cycle inheritance* if by blocking some newly added tasks and hiding others one cannot distinguish between them. All workflow processes shown in Figure 25 are subclasses of workflow process (A) with respect to life-cycle inheritance.

In [6,14] 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 just mention the four inheritance preserving transformation rules presented in [14]:

- *PT*
  Transformation rule *PT* preserves protocol inheritance and life-cycle inheritance. PT extends the superclass with new alternatives. In the resulting subclass there are alternative routes containing new tasks. Workflow process (A) shown in Figure 25 can be extended to workflow process (C) using this rule. However, rule PT allows for much more complex extensions involving the introduction of new alternative subflows containing many tasks and routing structures.
- *PP*
  Transformation rule *PP* preserves all four forms of inheritance, i.e., protocol/projection, projection, protocol, and life-cycle inheritance. Rule PP introduces new tasks which only postpone behavior. Workflow process (B) shown in Figure 25 can be constructed from (A) by applying this rule; task *check* only postpones the execution of *handle*.
- *PJ*
  Transformation rule *PJ* preserves projection inheritance and life-cycle inheritance. Rule PJ inserts new tasks in-between existing tasks. Workflow process (A) shown in Figure 25 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.
- *PJ3*
  Transformation rule *PJ3* preserves projection inheritance and life-cycle inheritance. Rule PJ3 adds parallel behavior. Workflow process (A) shown in Figure 25 can be extended to workflow process (D) using this rule.

The rules correspond to design constructs that are often used in practice, namely choice, iteration, sequential composition, and parallel composition. If the designer sticks to these rules, inheritance is guaranteed!

The reason we introduced the four inheritance preserving transformation rules is the following:

> *If all children of a process family are constructed from a minimal representative by using the four inheritance preserving transformation rules, then it is possible to generate suitable generalization (gen_i), specialization (spe_i) and management-information (man_i) functions automatically.*

This means that if the minimal representative is used as a template which is extended by applying the rules PT, PP, PJ, and PJ3, then the generalization, specialization, and management-information functions can be constructed automatically. Moreover, these functions yield a mapping such that the problems indicated before do not occur, i.e., every mapping yields a state which is as close to the real state as possible, and deadlocks, livelocks and other anomalies are avoided. The functions also satisfy the constraints stated in Section 5 (i.e., no Z-structures). The generalization and specialization functions are total, i.e., the case is transferred the moment the change occurs. As a result, the generalization functions *(gen_i)* and management-information functions *(man_i)* are identical.

Consider for example the five workflow processes in Figure 25. Suppose that workflow process (A) is the minimal representative and each of the five processes (i.e., including (A)) is a variant, i.e., a member of the process family having (A) as a minimal representative. Let gen_D, spe_D, and man_D be the generalization, specialization, and management-information function corresponding to variant (D) which are constructed using the inheritance preserving transformation rules. Functions gen_D and man_D both map the state with a token in *p1* and *p3* (i.e., only transition *register* was executed in workflow process (D)) onto the state with a token in *p1*. Function spe_D maps the state with a token in *p2* (i.e., the state just before executing *archive* in workflow process (A)) onto the state with a token in *p2* and either *p3* (conservative approach) or *p4* (progressive approach). Let gen_E, spe_E, and man_E be the generalization, specialization, and management-information function corresponding to variant (E). Functions gen_E and man_E both map the state with a token in *p3* onto the state with a token in *p2*. Function spe_E maps the state with a token in *p2* onto the state with a token in *p2*. For the simple extensions shown in Figure 25 the results may seem trivial. However, note that we can construct these functions for any extension which can be described as a sequence of the four transformation rules. Since the rules correspond to design constructs encountered in practice (choice, iteration, sequential composition, and parallel composition), the results are meaningful and far from trivial.

If every variant is a subclass of the minimal representative constructed using the four inheritance preserving transformation rules, then the transfer from one variant to another does not cause any problems, i.e., every case can be transferred without any delay and without introducing anomalies such as deadlocks, livelocks, unintended skipping of tasks, unnecessary multiple executions of common tasks, etc. Consider for example a transfer from workflow process (E) to workflow process (D). If the case is

in the state corresponding to *p3*, then the case is first mapped onto the minimal representative (A) using gen_E. The transient state of the case in workflow process (A) is the state with a token in place *p2*. From this transient state, the case is transferred to workflow process (D) using spe_D. The resulting state is a token in place *p2* and a token in either *p3* (conservative approach) or *p4* (progressive approach). Such dynamic changes can be handled automatically, e.g., the functions gen_E and spe_D can be computed based on the inheritance preserving transformation rules.

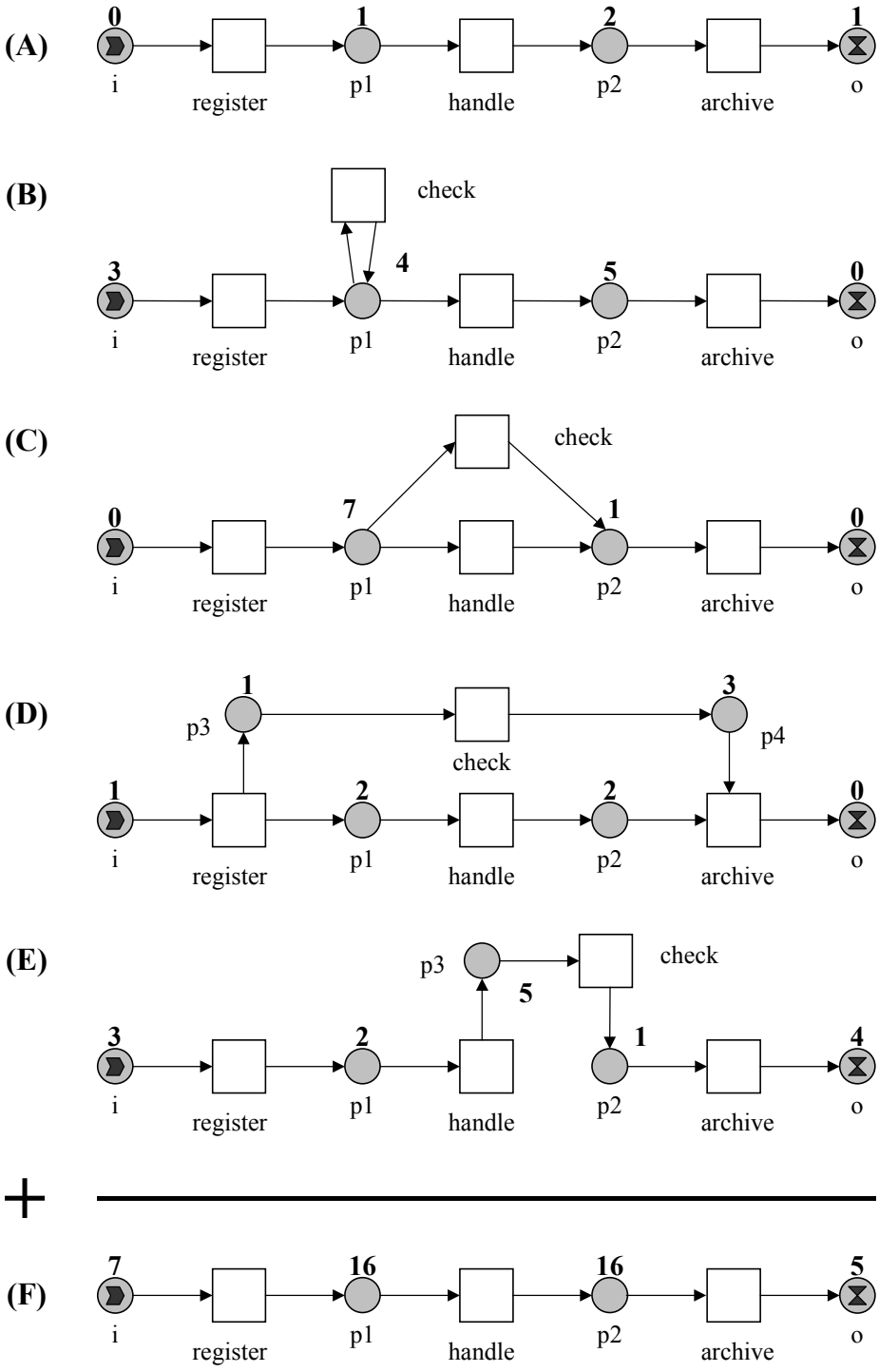

Figure 26: Aggregated management information mapped onto the minimal representative of the five variants.

Figure 26 shows five variants as described earlier augmented with numbers indicating the distribution of in total 44 cases. Workflow process (A) holds 4 cases, (B) holds 12 cases, (C) holds 8 cases, (D) holds 5 cases, and (E) holds 15 cases. Consider for example the five cases in variant (D). One case is still in the initial state. The remaining four are in-between *register* and *archive*; three of them have been checked and two of them have been handled. The minimal representative is the purely sequential process (i.e., workflow process (A)). Using the automatically constructed functions man_A, man_B, man_C, man_D, and man_E the 44 cases can be mapped onto the minimal representative; workflow process (F) shows the aggregated management information mapped onto this minimal representative. Seven cases are in the initial state (7=0+3+0+1+3), sixteen are in the state corresponding to *p1* (16=1+4+7+2+2), sixteen are in *p2* (16=2+5+1+2+6), and five are in the final state (5=1+0+0+0+4). Note that six of the fifteen cases in variant (E) are mapped onto *p2*.

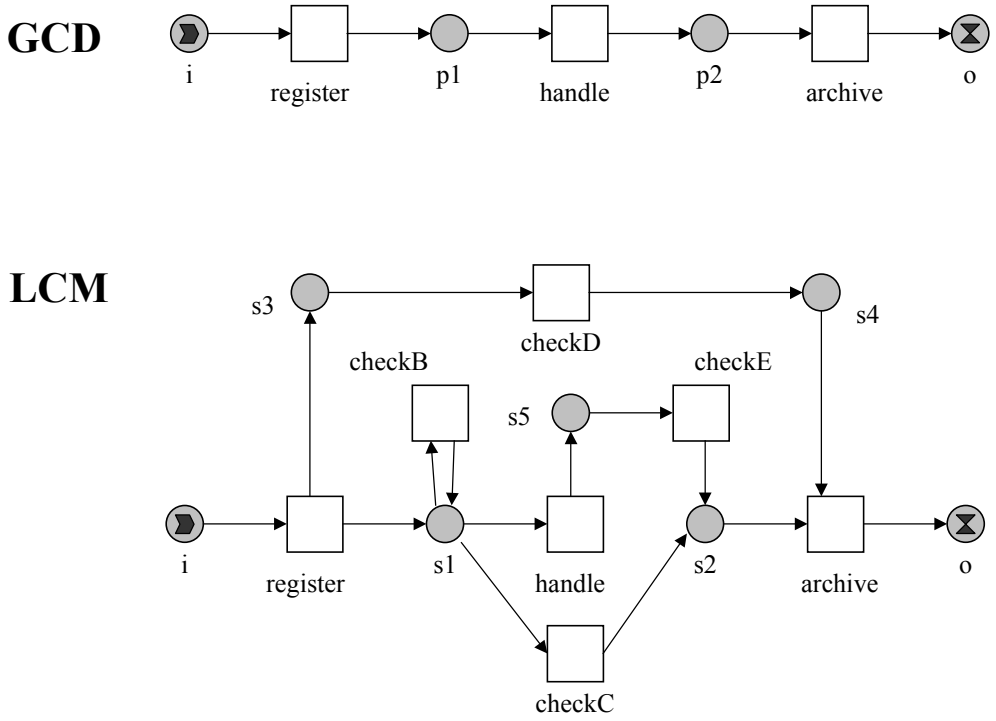

Figure 27: The GCD and the LCM of the five variants shown in Figure 25.

These examples show that the concept of a minimal representative can be made operational using the four inheritance preserving transformation rules. In fact, it is even possible to concretize the notions of GCD and LCM. The *Greatest Common Denominator (GCD)* of a set of variants is the "largest" workflow process such that every variant is a subclass with respect to life-cycle inheritance, i.e., it is the largest common part where all variant agree on. "Largest" is defined with respect to life-cycle inheritance, i.e., there is no other process which is a subclass of the GCD and a superclass of all variants with respect to life-cycle inheritance. Clearly workflow process (F) is the GCD of the five variants shown in Figure 26. The *Least Common Multiple* (LCM) of a set of variants is the "smallest" workflow process such that every variant is a superclass of this workflow process with respect to life-cycle inheritance, i.e., it is the smallest workflow process which captures all possible behaviors. The

LCM is the smallest workflow process such that each variant can be constructed by blocking and hiding the appropriate tasks in the LCM. Note that the LCM is a subclass of each of the variants and that life-cycle inheritance is used to compare processes. Figure 27 shows the LCM of the five workflow processes depicted in Figure 25. Some of the tasks have been renamed to avoid name classes. The mapping of the cases shown in Figure 26 onto this LCM is straightforward and can be done automatically, e.g., there are eleven cases in the state corresponding to place *s2* (11=2+5+1+2+1). A more detailed discussion of GCD and LCM is outside the scope of this paper. For a formal definition of GCD and LCM the reader is referred to [7].

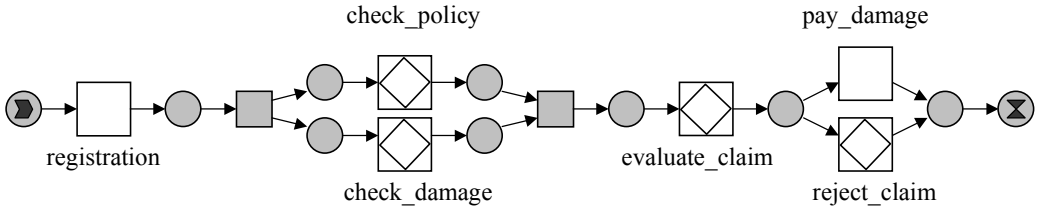

Figure 28: The revised non-atomic concrete process *handle_insurance_claim*.

To illustrate to application of the inheritance-based techniques we return to our process *handle_insurance_claim*. Figure 28 shows a version of the *handle_insurance_claim* where *check_policy* is no a concrete process but a generic one. Figure 29 shows the inheritance diagram for *check_policy*. Each of the children is a non-atomic concrete process. The process *minimal_policy_check* is the minimal representative and corresponds to the routing diagram given in Section 3.2. The other three children are augmented with an expression which makes sure that the configuration parameter *config_par_it* is set to the proper value. This configuration parameter can be used when instantiating other generic processes, e.g., the instantiation of generic process *check_damage*.

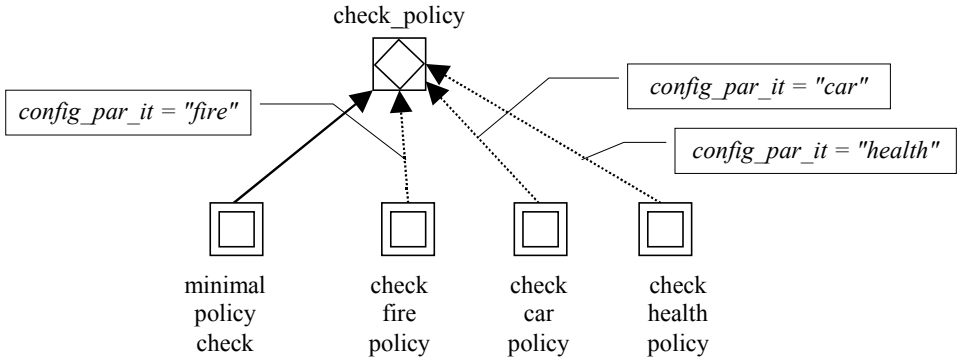

Figure 29: An inheritance diagram for the generic process *check_damage*.

Figure 30 shows the definition of each of the four non-atomic concrete processes mentioned in the inheritance diagram for *check_policy*. The minimal representative *minimal_policy_check* is chosen appropriately since each of the children of *check_policy* is a subclass of *minimal_policy_check*. In fact, *minimal_policy_check* is the GCD of the three other members. Since each of the variants is a subclass of *minimal_policy_check*, we can automatically construct the generalization, specialization, and management-information functions. Figure 30 shows the number

of cases in each variant/state combination, e.g., three claims for damage due to fire are in a state in-between querying the police records and assessing the risk. By using the automatically derived management-information functions, each case can be mapped onto the minimal representative. In this case the mapping is rather trivial: as Figure 30 shows 24 cases are in a state in-between querying policy data and querying payments data, 2 cases are in the beginning of the subprocess, and 3 cases have just completed both querying tasks.

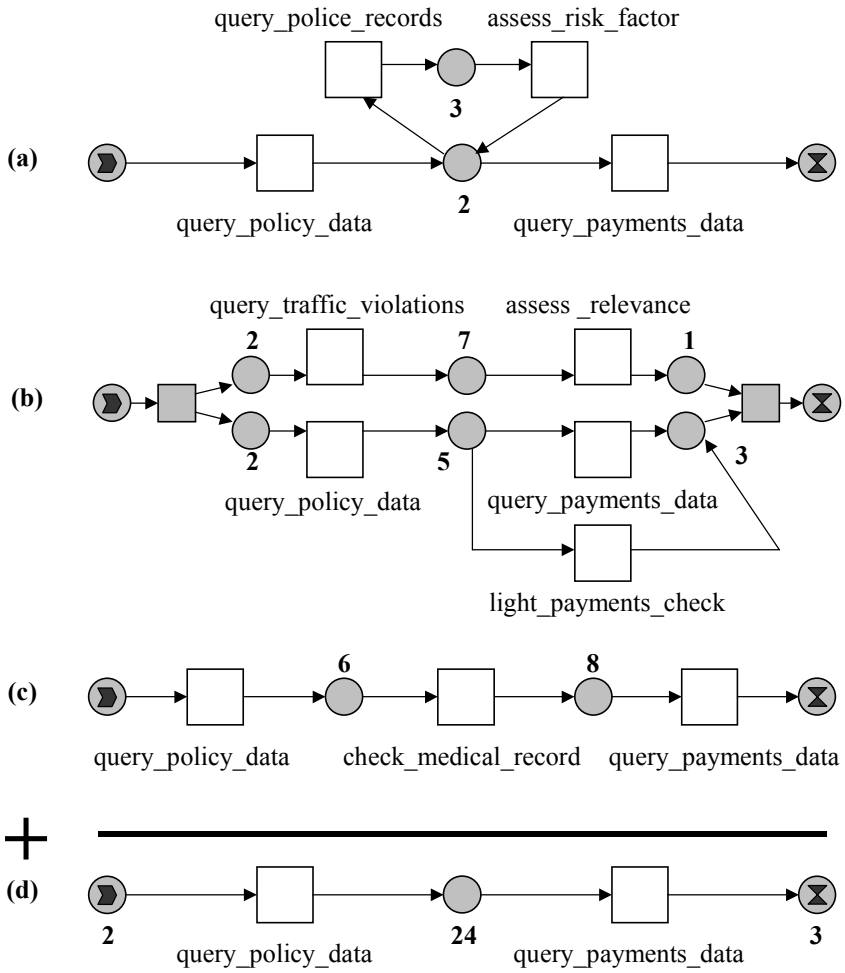

Figure 30: The four members of the process family *check_policy*: (a) *check_fire_policy*, (b) *check_car_policy*, (c) *check_health_policy*, and (d) *minimal_policy_check*.

As Figure 30 illustrates the inheritance concepts can be used to derive a minimal representative and all management information can be mapped onto the common superclass. The same concepts can be used to support dynamic change. First, based on the subclass-superclass relationship between the old process and the minimal representative, a generalization function can be derived which maps the state of a case in the old process onto the minimal representative. Then, the specialization function, derived from the superclass-subclass relationship between the minimal representative and the new process, can be used to migrate the case to the new process.

8 Related work

In the last decade, there have been numerous papers addressing workflow management and workflow management systems [33,38,39], but relatively little work is devoted to the problems related to change. A topic related to change is the handling of exceptions. There are several papers addressing exception handling, in particular in the context of transactional workflows [18,19,26,36,44]. However, these papers typically address the handling of failures and other undesirable events rather than the deliberate change of a workflow process, e.g., evolutionary change, logical anomalies such as the dynamic change problem, and extracting useful management information are topics not addressed in these papers. The first paper to address the problems related to dynamic change was the paper by Ellis, Keddara, and Rozenberg in 1995 [20]. This paper justifies research in this area by providing several examples of "dynamic bugs", i.e., errors resulting from change, and proposes a solution based on the notion of "change regions". A change region is the part of the process affected directly or indirectly by the change, i.e., the segment of the workflow process that may cause problems. For each change region, two versions are maintained in parallel: the old one and the new one. New cases, i.e., cases entering the change region, are handled according to the new version. Cases already populating the change region are handled the old way. Eventually, the old version of a change region becomes inactive (because all old cases have been handled) and can be removed. This approach has the drawback that the process definition can become very complex (the process definition contains both old and new versions of change regions). However, a more serious drawback is the fact that the change regions are identified manually and there is little support for the transfer of cases. In [21,34] the authors improve their approach by introducing jumpers. A jumper moves a case from the old workflow to the new workflow and if for a state no jumper is available, the jump is postponed. Again, the authors do not give a concrete technique for the transfer of cases, i.e., jumpers are added manually. Agostini and De Michelis [13] propose a technique for the automatic transfer of cases from the old process to the new process and also give criteria for determining whether a jump is possible. Unfortunately, the approach only works for a restricted class of workflows, e.g., the workflow models has to be acyclic (i.e., iteration is modeled through linear jumps). The authors claim that these restrictions are reasonable because one could consisder iteration as an exception. Klingemann et al. [25,37] propose and mixture of so-called mandatory elements (i.e., typical constructs such as sequence, OR-split, AND-split, OR-join, AND-join) and flexible elements. Examples of flexible elements are alternative activities (run-time binding), non-vital activities (elements that can be skipped), and optional execution orders (suggested but not enforced ordering). Casati, Ceri and Pernici [17] 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 (comparable to the approach using change regions). Reichert and Dadam [41] use a similar approach without addressing for example the problem identified in Figure 1. Voorhoeve and Van der Aalst [49,50] also propose a fixed set of transformation rules to support dynamic change. However, the drawback of using transformation rules is that only local changes are considered and the rules provided so far are far from being complete. Moreover, valuable information is lost during the application of a series of transformation rules. None of the above papers uses a notion of inheritance or generic processes. Moreover, the issue of extracting management information is only mentioned in [49,50]. The other papers mentioned do not address this issue.

The approach in this paper is inspired by the work on schema evolution [16] and generic/variant bills-of-material [23,29,46,47], and builds on previous work conducted by members of the SMIS group [1,4,6,14]. Preliminary results have been presented by the author in [5]. Compared to [5] this paper contains much more details, a classification of change, several examples, and the link with the work on inheritance of dynamic behavior [6,14]. The inheritance notions, transformation rules, transfer rules, and the notions of GCD and LCM provide concrete mechanisms for defining a minimal representative, handling dynamic change, and generating management information, and are therefore crucial for the applicability of the approach. For a formalization of the inheritance concepts we refer to two technical reports [7,15] containing detailed proofs of the statements made in this paper. Moreover, the four inheritance notions can be verified with Woflan [48]. Woflan is our workflow verification tool and can interface with several workflow products including COSA, Staffware, Protos, and Meteor.

9 Conclusion

This paper tackled two notorious problems related to adaptive workflow using generic process models. The approach is inspired by the work on product configuration (generic bills-of-material). The generic process model extends the classical workflow models, primarily based on routing diagrams, with inheritance diagrams. This allows for the specification of process families composed of variants. It also provides the designer with two navigation dimensions: (1) the is-part-of/contains dimension and (2) the generalizes/specializes dimension, and stimulates reuse. Based on this model the problems related to (1) providing *management information* at the right aggregation level and (2) supporting *dynamic change* (i.e., migrating cases from an old to a new workflow) have been addressed. As it turns out, the generic process model with a minimal representative for each process family gives a handle to deal with these problems. Although the diagrams shown in this paper use a Petri-net-like notation, the concepts and ideas are independent of the process modeling technique chosen. Therefore, it is, in principle, possible to add the notions presented in this paper to most of the workflow management systems available today. However, the generality of the approach also indicates that many problems are still open. For example, how to construct a good a minimal representative and the corresponding specialization (spe_i), generalization (gen_i) and management functions (man_i)? To answer these questions, we proposed an approach based on Petri nets and advanced inheritance concepts. By using recent results on inheritance of dynamic behavior [6,7,14] we showed that the corresponding specialization, generalization, and management functions can be obtained automatically if the minimal representative is a superclass of its fellow children.

References

1. W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407-426. Springer-Verlag, Berlin, 1997.
2. 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., editor, *Information and Process Integration in*

Enterprises: Rethinking documents, The Kluwer International Series in Engineering and Computer Science, pages 161-182. Kluwer Academic Publishers, Norwell, 1998.

3. W.M.P. van der Aalst. Finding Errors in the Design of a Workflow Process: A Petri-net-based Approach. 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)*, volume 98/7 of *Computing Science Reports*, pages 60-81. Eindhoven University of Technology, Eindhoven, 1998.
4. 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.
5. W.M.P. van der Aalst. Generic Workflow Models: How to Handle Dynamic Change and Capture Management Information. In M. Lenzerini and U. Dayal, editors, *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems (CoopIS'99)*, pages 115-126, Edinburgh, Scotland, September 1999. IEEE Computer Society Press.
6. W.M.P. van der Aalst and T. Basten. Life-cycle Inheritance: A Petri-net-based approach. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 62-81. Springer-Verlag, Berlin, 1997.
7. W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An approach to tackling problems related to change. Computing Science Reports 99/06, Eindhoven University of Technology, Eindhoven, 1999. To appear in *Theoretical Computer Science*.
8. W.M.P. van der Aalst, T. Basten, H.W.M. Verbeek, P.A.C. 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*, Setubal, Portugal, pages 353-360, 1998.
9. W.M.P. van der Aalst and K.M. van Hee. Business Process Redesign: A Petri-net-based approach. *Computers in Industry*, 29(1-2):15-26, 1996.
10. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Modellen, Methoden en Systemen (in Dutch)*. Academic Service, Schoonhoven, 1997.
11. W.M.P. van der Aalst and S. Jablonski, editors. *Flexible Workflow Technology Driving the Networked Economy*, Special Issue of the *International Journal of Computer Systems, Science, and Engineering*, volume 15, number 5, 2000.
12. 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*). UNINOVA, Lisbon, June 1998.
13. 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)*, volume 98/7 of *Computing Science Reports*, pages 146-164. Eindhoven University of Technology, Eindhoven, 1998.
14. 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.
15. T. Basten and W.M.P. van der Aalst. Inheritance of Behavior. Computing Science Reports 99/17, Eindhoven University of Technology, Eindhoven, 1999. To appear in the *Journal of Logic and Algebraic Programming*.
16. E. Bertino, E. Ferrari, and V. Atluri. Object-Oriented Database Systems: Concepts and Architecture. Addison-Wesley, 1993.
17. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. *Data and Knowledge Engineering*, 24(3):211-238, 1998.
18. J. Eder and W. Leibhart, The Workflow Activity Model WAMO. *Proceedings of the Third International Conference on Cooperative Systems (CoopIS'95)*, Vienna, Austria, May 1995.
19. J. Eder and W. Leibhart, Contributions to exception handling in workflow management. In O. Burkes, J. Eder, and S. Salza, editors, *Proceedings of the Sixth International Conference on Extending Database Technology*, pages 3-10. Valencia, Spain, March 1998.
20. C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In N. Comstock and C. Ellis, editors, *Conf. on Organizational Computing Systems*, pages 10 - 21. ACM SIGOIS, ACM, Aug 1995. Milpitas, CA.

21. 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)*, volume 98/7 of *Computing Science Reports*, pages 109-128. Eindhoven University of Technology, Eindhoven, 1998.
22. C.A. Ellis and G.J. Nutt. Modeling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 1-16. Springer-Verlag, Berlin, 1993.
23. F. Erens, A. MacKay, and R. Sulonen. Product modelling using multiple levels of abstraction - instances as types. *Computers in Industry*, 24(1):17-28, 1994.
24. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics In G.X. Ritter, editor, *Information Processing 89: Proceedings of the IFIP 11th. World Computer Congress*, pages 613-618, San Francisco, CA, USA, August/September 1989. Elsevier Science Publishers B.V., North-Holland, 1989.
25. P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. CrossFlow: Cross-Organizational Workflow Management in Dynamic Virtual Enterprises. *International Journal of Computer Systems, Science, and Engineering*, 15(5):277-290, 2000.
26. C. Hagen and G. Alonso. Flexible Exception Handling in the OPERA Process Support System. In proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS), Amsterdam, The Netherlands, 1998.
27. 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, July 1998.
28. K. Hayes and K. Lavery. Workflow management software: the business opportunity. Technical report, Ovum Ltd, London, 1991.
29. H.M.H. Hegge. Intelligent Product Family Descriptions for Business Applications. PhD thesis, Eindhoven University of Technology, Eindhoven, 1995.
30. 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, 1998.
31. A.H.M. ter Hofstede, M.E. Orlowska, and J. Rajapakse. Verification Problems in Conceptual Workflow Specifications. *Data and Knowledge Engineering*, 24(3):239-256, 1998.
32. InConcert. *InConcert Process Designer's Guide*. InConcert Inc, Cambridge, Massachusetts, 1997.
33. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation* International Thomson Computer Press, 1996.
34. K. Keddara. Dynamic Evolution of Workflow Systems. PhD thesis, University of Colorado, Department of Computer Science, Boulder, USA, 1999.
35. E. Kindler. Database Theory - Petri Net Theory - Workflow Theory. Informatikberichte 102, Humboldt-Universität zu Berlin, Berlin, 1998.
36. M. Klein, C. Dellarocas, and A. Bernstein, editors. *Proceedings of the CSCW-98 Workshop Towards Adaptive Workflow Systems*, Seattle, Nov. 1998.
37. J. Klingemann. Controled Flexibility in Workflow Management. In proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE'00), Stockholm, Sweden, pages 126-141, 2000.
38. T.M. Koulopoulos. *The Workflow Imperative*. Van Nostrand Reinhold, New York, 1995.
39. P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York, 1997.
40. T. Malone, W. Crowston, J. Lee, B. Pentland, and et. al. Tools for inventing organizations: Toward a handbook for organizational processes. *Management Science*, 1998 (to appear).
41. M. Reichert and P. Dadam. ADEPTflex: Supporting dynamic changes of workflow without loosing control. *Journal of Intelligent Information Systems*, 10(2):93-129, 1998.
42. T. Schäl. *Workflow Management for Process Organisations*, volume 1096 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996.

43. A. Sheth. From Contemporary Workflow Process Automation to Dynamic Work Activity Coordination and Collaboration. *Siggroup Bulletin*, 18(3):17-20, 1997.
44. D.M. Strong and S.M. Miller. Exceptions and exception handling in computerized information processes. *ACM Transactions on Information Systems*, 13(2):206-233, 1995.
45. R. Valette. Analysis of Petri Nets by Stepwise Refinements. *Journal of Computer and System Sciences*, 18:35-46, 1979.
46. E.A. van Veen and J.C. Wortmann. Generative bill of matarial processing systems. *Production Planning and Control*, 3(3):314-326, 1992.
47. E.A. van Veen and J.C. Wortmann. New developments in generative bom processing systems. *Production Planning and Control*, 3(3):327-335, 1992.
48. E. Verbeek. Woflan home page. http://www.tm.tue.nl/it/woflan, 1999.
49. M. Voorhoeve and W.M.P. van der Aalst. Conservative Adaptation of Workflow. In M. Wolf and U. Reimer, editors, *Proceedings of the International Conference on Practical Aspects of Knowledge Management (PAKM'96), Workshop on Adaptive Workflow*, pages 1-15, Basel, Switzerland, Oct 1996.
50. 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, Sept 1997.
51. WFMC. Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011). Technical report, Workflow Management Coalition, Brussels, 1996.
52. 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, Oct 1996.

Appendix: Petri nets

The results in this paper are not specific for the Petri-net formalism. However, to understand the diagrams some basic knowledge of Petri nets is required. In this appendix, we introduce the formalism for readers not familiar with Petri nets. We also provide some background information with respect to the modeling and verification of workflows.

Because processes are the dominant factor in workflow management, it is important to use an established framework for modeling and analyzing workflow processes. In this paper, we use a framework based on Petri nets to illustrate the concepts. Petri nets are a well-founded process modeling technique. The classical Petri net was invented by Carl Adam Petri in the sixties. Since then Petri nets have been used to model and analyze all kinds of processes with applications ranging from protocols, hardware and embedded systems to flexible manufacturing systems, user interaction and business processes. There are several reasons for using Petri nets for workflow modeling: their formal semantics, graphical nature, expressiveness, analysis techniques, and tools provide a framework for modeling and analyzing workflow processes [2].

A *Petri net* is a network composed of squares and circles. The squares are called *transitions* and correspond to tasks that need to be executed. The circles are used to represent the state of a workflow and are called *places*. The arrows between places and transitions are used to specify causal relations. A place p is called an input place of a transition t iff there exists a directed arc from p to t. Place p is called an output place of transition t iff there exists a directed arc from t to p. At any time a place contains zero or more tokens, drawn as black dots. The state of the net, often referred

to as marking, is the distribution of tokens over places. The number of tokens may change during the execution of the net. Transitions are the active components in a Petri net: they change the state of the net according to the following firing rule:

1. A transition *t* is said to be *enabled* iff each input place *p* of *t* contains at least one token.
2. An enabled transition may *fire*. If transition *t* fires, then *t* consumes one token from each input place *p* of *t* and produces one token for each output place *p* of *t*.

By using this rule it is possible to determine which transitions can fire and in what order.

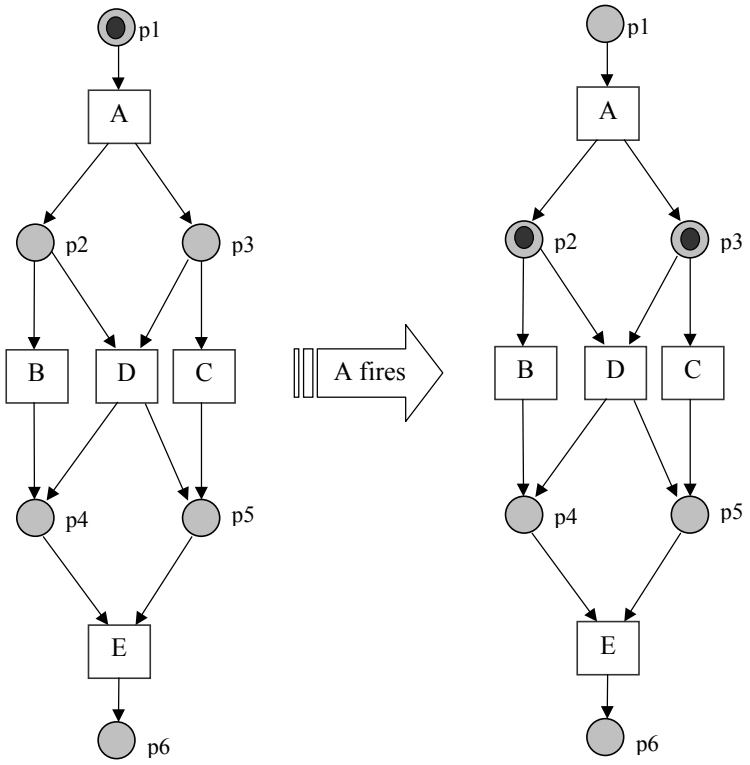

Figure 31: The result of firing the enabled transition A.

Figure 31 shows a Petri net before (left) and after (right) the firing of a transition. In the Petri net before the firing, only place *p1* contains a token and transition *A* is the only enabled transition. Therefore, *A* will fire and consume one token and produce two tokens resulting in the state shown in Figure 31. In the resulting state, three transitions are enabled (*B*, *C*, and *D*). Either *B* and *C* fire (in parallel or in any order) or *D* fires. Finally, *E* will fire resulting in the state with just one token in *p6*.

A Petri net which models the process aspect of a workflow, is called a *WorkFlow net* (WF-net). It should be noted that a WF-net specifies the dynamic behavior of a single case in isolation. A WF-net is a Petri net with one source place (i.e., a place with no ingoing arcs) and one sink place (i.e., a place with no ingoing arcs), and every node (i.e., a place or a transition) is on a path from the source place to the sink place [4]. A WF-net has one input place (source) and one output place (sink) because any case handled by the procedure represented by the WF-net is created the moment it enters the workflow management system and is deleted once it is completely handled by the workflow management system, i.e., the WF-net specifies the life-cycle of a case. Moreover, any node should be on a path from the input place to the output place. This

requirement has been added to avoid ‘dangling tasks and/or conditions’, i.e., tasks and conditions which do not contribute to the processing of cases. The Petri net shown in Figure 31 is clearly a WF-net. A WF-net is *sound* if and only if it satisfies the following requirement. *For any case, the procedure will terminate eventually and the moment the procedure terminates there is a token in the output place and all the other places are empty.* Moreover, there should be no dead tasks, i.e., it should be possible to execute an arbitrary task by following the appropriate route though the WF-net. A formal definition of soundness is given in [1,4]. It is easy to verify that the WF-net net shown in Figure 31 is sound. However, for the complex workflow process definitions encountered in practice (with up to 100 tasks), it is far from trivial to decide soundness. In [1] a decision procedure is given to decide soundness. This procedure uses state-of-the-art Petri-net-based analysis techniques. In fact, it uses the fact that soundness corresponds to two well-known properties: liveness and boundedness.

Acknowledgements

The author would like to thank all (former) the members of the SMIS group, in particular Twan Basten for his excellent work on inheritance of dynamic behavior and Eric Verbeek for the development of Woflan. The author would also like to thank the reviewers for the comments that helped improving this paper.

About the author

Wil van der Aalst is a full professor of Information Systems and head of the Department of Information and Technology of the Faculty of Technology Management of Eindhoven University of Technology. He is also a part-time full professor at the Computing Science department of the same university and has been working as a part-time consultant for Bakkenist for several years. Wil van der Aalst also directs the Eindhoven Digital Laboratory for Business Processes (EDL-BP) and is a fellow and management team member of the research institute BETA. His research interests are information systems, simulation, Petri nets, process models, workflow management systems, verification techniques, enterprise resource planning systems, computer supported cooperative work, and interorganizational business processes.