Generic Workflow Models: How to Handle Dynamic Change and Capture Management Information?

W.M.P. van der Aalst†
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
wsinwa@win.tue.nl

Abstract

Traditionally, workflow management systems are used to support static processes, i.e., processes which do not change frequently. This has limited the scope of workflow management. Moreover, the networked economy of the new millennium requires workflow management systems which are able to deal with dynamically changing workflow processes. 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. It is a first step in the direction of truly flexible workflow management systems and provides a handle to solve the two problems mentioned.

1. Introduction

The new millennium is characterized by an increasing number of business processes subject to continuous change. Organizations are challenged to bring ideas and concepts to products and services in an ever-increasing pace. Companies distributed by space, time and capabilities come together to deliver products and solutions for which there is any need in the global marketplace. The trends for virtual corporations and e-commerce, and increasing global networking of economies are real and will accelerate. As a result, more and more workflow processes are subject to continuous change. At the moment, there are many 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 [4,7,10,11,12,15,18,26,28,32]. 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 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 [4]: (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.

For the past five years the author has been active as a consultant for Bakkenist Management Consultants in the area of workflow management. In this period Bakkenist has supported numerous workflow projects for large financial institutions and the Dutch government (see http://www.bakkenist.nl). Based on practical experiences while selecting, testing, and configuring various workflow management systems, the author was confronted with the problem of change.

This paper presents an approach to tackle the problem of change. This approach is inspired by the techniques used in *product configuration* [30]. As factories have to manufacture more and more customer specific products,

† Part of this work was done at AIFB (University of Karlsruhe, Germany) and LSDIS (Univeristy of Georgia, USA) during a sabbatical leave.

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* [14,17,30,31] 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*.

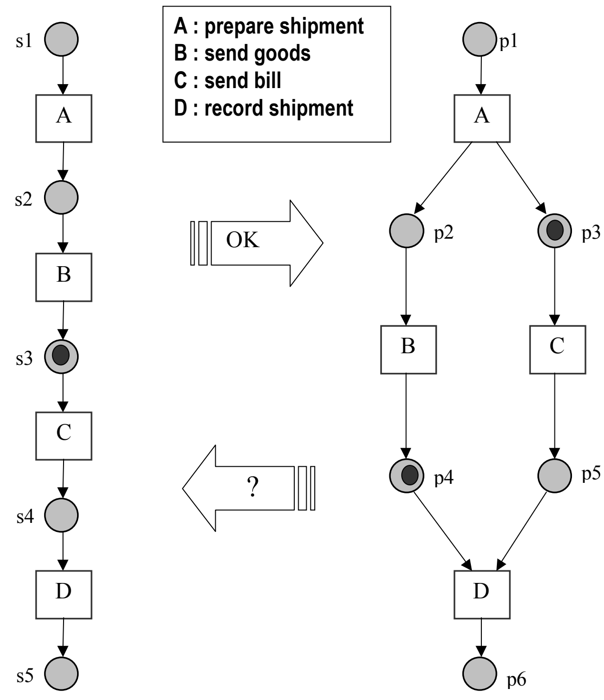

Figure 1: The dynamic change problem.

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* [32,33], and (2) *dynamic change* [7,11,12]. 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 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 starts in the state with one token in *s1* and has five possible states. 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

[1] In this paper, we use Petri nets to illustrate the main concepts. 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 of 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*.

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.

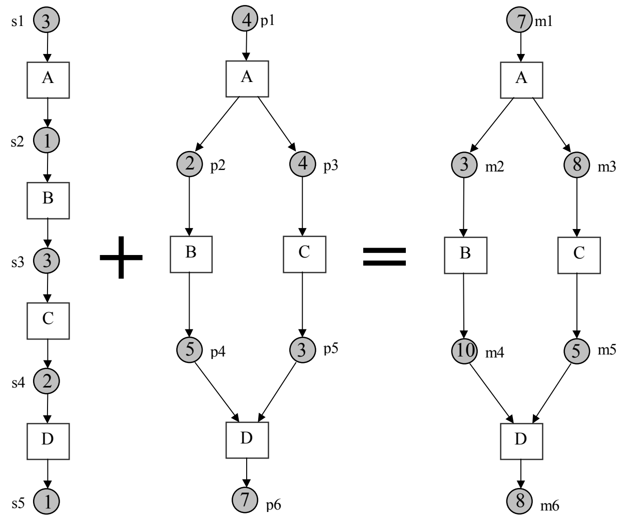

Figure 2: Aggregated management information.

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 divisor' 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*, it is quite straightforward to verify that the numbers are correct. The number of tokens in place *m3* corresponds to the number of cases in-between *A* and *C*. In the sequential process there are 1+3=4 cases in-between *A* and *C*. In the parallel process there are also 4 cases in-between *A* and *C*, which brings the total to 8. For this small example it may seem trivial to obtain this information. However, in general there are many variants 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.

This paper extends the results presented in [2] by addressing the problems illustrated by Figure 1 and Figure 2. The remainder 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. Finally, 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.

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 [24,34]). 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, i.e., the domain 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 [1,10,12,19,24,27,34] 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* [7,10,11,12,15,18,28,32]. 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 [6,22,35]. 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 [4]:

- ❖ *Ad-hoc change*: Changes occurring on an individual basis, i.e., only a single case (or a limited set of cases) 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 factor. A typical dilemma related to ad-hoc change is the problem to decide what kinds of changes are allowed and the fact that it is impossible to foresee all possible changes. 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, i.e., 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 adaptive workflow. Note that the term *exception handling* does not appear in the classification. 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.

The classification just given 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). Examples of cases are tax declarations, complaints, job applications, credit card payments, and insurance claims. A *task* 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. A *concrete process* is either a task or a non-atomic concrete process, i.e., it is 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 refer to both concrete tasks and non-atomic concrete processes at the same time. A *process node* is either a concrete process or a generic process. 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 of 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.

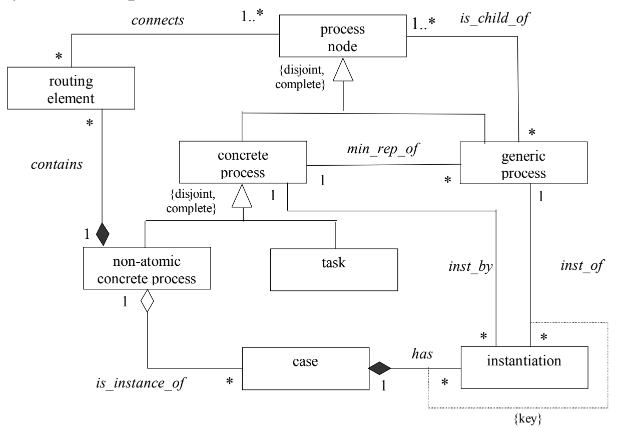

Figure 3: Class diagram describing the relationships between the main concepts used in this paper.

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 [34]. 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.*

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. None of these diagramming techniques supports generic processes. However, each of these diagramming techniques can be extended 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 workflow node.
4. *Routing elements*: Routing elements are added to specify which workflow 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.

Since any of the workflow management system available has some kind of diagramming technique, we simply use Petri-net like diagrams and formulations. In fact, we extend Petri-net-like routing diagrams [1,12] with generic processes. Since most of the readers are familiar with similar diagramming techniques, we do not go into details. The only relevant aspect is the addition of generic processes in the routing diagrams.

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 [14,17,30,31]. 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 4 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 [3,8]. 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 4. 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. [1]). 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 [30] for the generic BOM. Note that the use of inheritance diagrams is also advocated by other researchers such as Malone et al. [25]. However, these researchers do not tackle the problems related to change, i.e., dynamic change and management information.

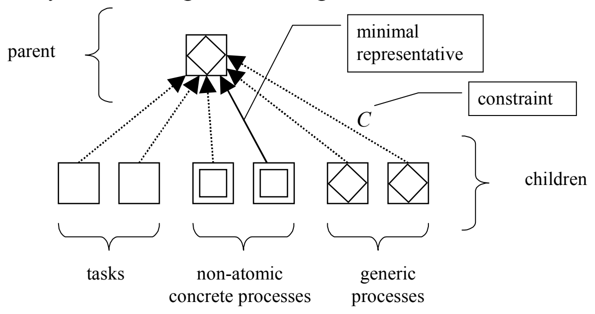

Figure 4: Symbols used in an inheritance diagram.

4. Dynamic change

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. [9]) 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.

Several researchers have worked on problems related to dynamic change. Ellis, Keddara and Rozenberg [11] propose a technique based on so-called "change regions", i.e., all parts of the workflow process that cause problems have two versions: the old one and the new one. This way, there is one version which covers the old and the new situation and changes affect cases as soon as possible. Parts of the workflow (i.e., change regions) become inactive after a while because all old cases have been handled. This approach has the drawback that the process definition can become very complex. 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 [12] 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 [6] 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., iteration can only be handled by ad-hoc jumps at runtime). Casati, Ceri and Pernici [10] 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[26] use a similar approach without addressing for example the problem identified in Figure 1. Voorhoeve and Van der Aalst [32,33] 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.

Independent of the approach used, the following two issues constitute a policy for dynamic change: (1) When to jump from the old process to the new process definition? and (2) 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". If a part is changed which does not correspond to a generic process, then a generic process is introduced. 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.

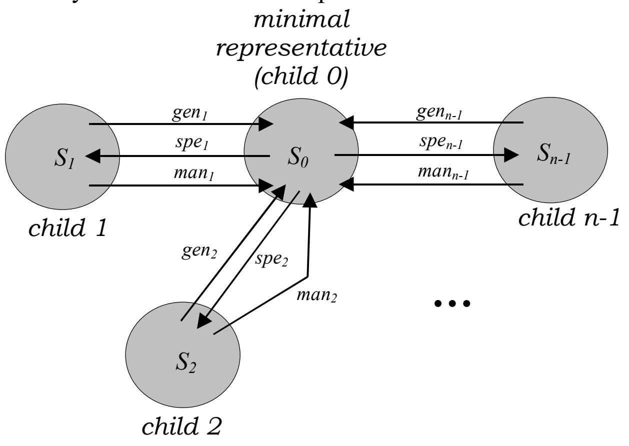

Figure 5: The relation between the state spaces of the minimal representative and its fellow children.

Figure 5 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 5 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.

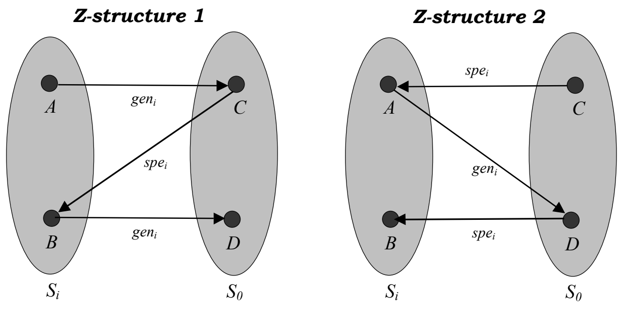

Figure 6: Two Z-structures that are not allowed.

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 6 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 6, 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., $S_i^J = \varnothing$. 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.

5. 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 divisor' (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.

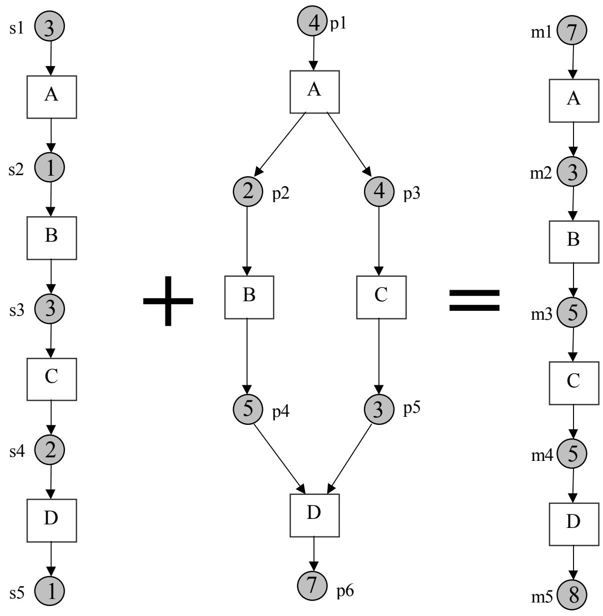

Figure 7: Aggregated management information mapped onto a sequential minimal representative.

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 $man_i \in S_i \to S_0$ 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 7 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 7 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, a badly chosen minimal representative will lead to misleading management information.

6. 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 (routing diagrams) and (2) the generalizes/specializes dimension (inheritance diagrams), 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 need to select a process modeling technique to reason about the dynamic properties of processes. Future research aims at answering these questions in a Petri-net-based setting using recent results on inheritance of dynamic behavior [3,5,7]. In [3,5,7] transformation rules are given that preserve certain inheritance relations. If such relations exist between the minimal representative and its fellow children, i.e., every child is a subclass of the minimal representative, then it should be possible to truly solve the problems discussed in this paper.

7. References

1 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.

2 W.M.P. van der Aalst. Flexible Workflow Management Systems: An Approach Based on Generic Process Models. To appear in the proceedings of DEXA'99, *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1999.

3 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.

4 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. 1998. In J. Filipe and J. Cordeiro, editors, *Proceedings of the first International Conference on Enterprise Information Systems*, pages 353-360, 1999.

5 W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. Technical report. Eindhoven University of Technology, Eindhoven, 1999.

6 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.

7 A. Agostini and G. De Michelis. Simple Workflow Models. In [6], pages 146-164.

8 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.

9 E. Bertino, E. Ferrari, and V. Atluri. Object-Oriented Database Systems: Concepts and Architecture. Addison-Wesley, 1993.

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

11 C. 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.

12 C.A. Ellis, K. Keddara, and J. Wainer. Modeling Workflow Dynamic Changes Using Timed Hybrid Flow Nets. In [6], pages 109-128

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

14 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.

15 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.

16 K. Hayes and K. Lavery. Workflow management software: the business opportunity. Technical report, Ovum Ltd, London, 1991.

17 H.M.H. Hegge. Intelligent Product Family Descriptions for Business Applications. PhD thesis, Eindhoven University of Technology, Eindhoven, 1995.

18 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.

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

20 E. Kindler. Database Theory - Petri Net Theory - Workflow Theory. Informatikberichte 102, Humboldt-Universität zu Berlin, Berlin, 1998.

21 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.

22 M. Klein, C. Dellarocas, and A. Bernstein, editors. *Proceedings of the CSCW-98 Workshop Towards Adaptive Workflow Systems*, Seattle, Nov. 1998.

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

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

25 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).

26 M. Reichert and P. Dadam. ADEPTflex: Supporting dynamic changes of workflow without loosing control. *Journal of Intelligent Information Systems*, 10(2):93-129, 1998.

27 T. Schäl. *Workflow Management for Process Organisations*, volume 1096 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996.

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

29 R. Valette. Analysis of Petri Nets by Stepwise Refinements. *Journal of Computer and System Sciences*, 18:35-46, 1979.

30 E.A. van Veen and J.C. Wortmann. Generative bill of matarial processing systems. *Production Planning and Control*, 3(3):314-326, 1992.

31 E.A. van Veen and J.C. Wortmann. New developments in generative bom processing systems. *Production Planning and Control*, 3(3):327-335, 1992.

32 M. Voorhoeve and W.M.P. van der Aalst. Conservative Adaptation of Workflow. In [35], pages 1-15.

33 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.

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

35 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.