Beyond Workflow Management: Product-Driven Case Handling

W.M.P. van der Aalst
Department 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

P.J.S. Berens
Pallas Athena
P.O. Box 747
NL 7300 AS, Apeldoorn
The Netherlands
paul.berens@pallas-athena.com

ABSTRACT

In the last decade, workflow technology has become one of the building blocks for realizing enterprise information systems. Unfortunately, the application of contemporary workflow management systems is limited to well-defined and well-controlled environments. In practice, workflow technology often fails because of limited flexibility. We advocate a paradigm shift to overcome this problem: Workflows should not be driven by pre-specified control-flows but by the products they generate. This paper presents the software package FLOWer which fully supports this paradigm shift.

Categories and Subject Descriptors

H.4.1 [**Information Systems Applications**]: Office Automation - *Workflow management.*

General Terms

Management, Design, Human Factors, Languages, Theory, Verification.

Keywords

Workflow management, case handling, workflow management systems, product-driven design, FLOW*er*.

1. INTRODUCTION

In the last three decades, there have been many attempts to build information systems to support (business) processes. What was called 'office automation' in the mid-70s is nowadays called 'workflow management'. From a technological point of view, much progress has been made. Moreover, processes moved from second-class citizens to first-class citizens of the enterprise information system. The traditional data-oriented approaches have been replaced by process-oriented paradigms (e.g., BPR, CPI, etc.) [9]. Unfortunately, despite all the jubilant stories of suppliers, the application of workflow management (WFM) systems [12,15,18] is still limited. Where WFM is actually applied, it is clear that the majority of the supported processes are very simple. For more complex processes, WFM systems are either not applicable or require considerable modeling and implementation efforts. For non-trivial processes, the degree of customization is typically high. The process models driving the WFM system are either kept simple by removing all flexibility or are complex and non-transparent to address exceptions adequately.

If the process model is kept simple, only a more or less idealized version of the preferred process is supported. As a result, 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 WFM system quite frequently, the system is more a liability than an asset. If the process model attempts to capture all possible exceptions [20], the resulting model becomes too complex to manage and maintain.

Many authors have pointed out the importance of workflow flexibility, e.g., many workshops and special issues of journals have been devoted to the topic [3,5,13,14]. Few of the results reported are really applicable in a practical setting and the impact on contemporary WFM systems is limited. We argue that the core of the problem is the fact that routing is the only mechanism driving the workflow, i.e., work is moved from on worktray to another based on pre-specified causal relations. This causes the following problems:

- Work needs to be *straight jacketed into activities*. Although activities are considered to be atomic by the WFM system, they are not atomic for the user. Clustering atomic activities into workflow activities is required to distribute work. However, the actual work is done at a much more fine-grained level.
- Routing is used for both work *distribution* and *authorization*. As a result, only crude mechanisms can be used to align workflow and organization.
- By focusing on control flow the context, i.e., data related to the entire case and not just the activity, is moved to be background. Typically, such *context tunneling* results in errors and inefficiencies.
- Routing focuses on what *should* be done instead of what *can* be done. This push-oriented perspective results in rigid inflexible workflows.

To overcome these problems we propose a paradigm shift: The *case* and not the routing should drive the workflow. The case is

GROUP'01, Sept. 30-Oct. 3, 2001, Boulder, Colorado, USA.

the object which is being handled. One should think of the case as being the product which is 'manufactured' by executing the workflow process. The characteristics of the product should drive the workflow. Typically, the product is information, e.g., a decision based on various data. By focusing on the product characteristics, one can replace push-oriented routing from one worktray to another by pull-oriented mechanisms centered around the data objects relevant for a case.

In this paper, we propose *product-driven case handling* as the mechanism to support, distribute, and manage work processes. The focus is on work processes where traditional WFM systems based on a traditional flow-oriented production-line paradigm fall short. FLOW*er*, developed by Pallas Athena and based on practical experiences with contemporary WFM systems, fully supports this paradigm shift. The interested reader is invited to play with an on-line demo which shows the functionality of FLOW*er*: http://www.pallas-athena.com/.

2. ON THE RELATION BETWEEN MANUFACTURING AND WORKFLOW

From a logistical point of view, there are many similarities between administrative processes and production processes (cf. Platier [17]). Both kinds of processes focus on the routing of work (workflow) and the allocation of work to resources. In a production system, the products are physical objects and the principal resources are machines, robots, humans, conveyor belts and trucks. In an administrative process the products are often informational (e.g. documents) and most of the resources are human. Although there are many similarities, there are also some logistical aspects in which an administrative process differs from a typical manufacturing process [2]:

- *Making a copy is easy and cheap.* In contrast to making a copy of a product like a car, it is relatively easy to copy a piece of information (especially if it is in electronic form).
- *There are no real limitations with respect to the in-process inventory.* Informational products do not require much space and are easy to access (especially if they are stored in a database).
- *There are less requirements with respect to the order in which activities are executed.* Human resources are flexible and there are few technical constraints.
- *Quality is difficult to measure.* What is the quality of the decision to accept an insurance claim?
- *Quality of end-products may vary.* A manufacturer of cars has a minimal quality level that any product should satisfy. However, in an administrative process it might be attractive to skip certain checks to reduce the workload.
- *Transportation of electronic data is timeless.* In a network information travels at the speed of light.
- *Production to stock is seldom possible.* Every product is unique, therefore it is difficult to produce in advance. It is not possible to process an insurance claim before it arrives.
- *Loops or rework occurs* frequently in administrative processes, but are very seldom or even impossible in production processes. In addition the required behavior is very different. In an administrative process reuse of information is easy, but in a production process reuse requires disassembly and reassembly.
- *The customer (can) influence(s)* the handling in an administrative process whereas in a production process the influence is restricted to before the process starts.

Nevertheless, the two types of processes have a lot in common. Consider for example performance indicators such as throughput time, waiting time, service level and utilization. These performance indicators play a prominent part in both domains.

In manufacturing, the Bill-Of-Material (BOM) is used to drive the production process [16]. Consider for example Material Requirements Planning, often referred to as MPR-I, which determines the production schedule based on the ordered quantities, current stock, and the composition of product as specified in the BOM. Contemporary Enterprise Resource Planning (ERP) systems such as SAP R/3 also take resource availability into account and use more refined algorithms. Nevertheless, production is driven by the structure of the product and product design is driven by desirable characteristics of the manufacturing process.

For administrative processes generating information-intensive products such as mortgage loans, driving permits, customs declarations, salary payments, etc. the relation between the product and the process is seldom made explicit. Clearly, the interaction between the product and the process is in the back of everyone's mind. However, there is not a standardized way to describe the product and the workflow process is usually designed without proper consideration of the product. Consider for example the processing of insurance claims. The product is basically a decision: Either the claim is accepted (followed by a payment) or the claim is rejected. All kind of information elements may play a role in making this decision. One can think of these information elements as raw materials or subassemblies. The workflow process should manufacture the decision while taking criteria such as product quality, average flow time, service level, and handling costs into account.

Driven by management principles such as Business Process Reengineering (BPR) [9], the focus has shifted from data and organization to processes. The problem is not this focus but the way processes are designed [19]. Workflow designs are typically made by small groups of consultants, managers and specialists. As a result, the processes are incomplete, subjective, and at a too high level ("The devil is in the details").

What can we learn from all of this?

- *Lesson 1*: There are many differences between administrative processes and production processes. Therefore, one should not simply transfer concepts from manufacturing to workflow. The push-oriented nature of contemporary WFM systems is not applicable for highly-dynamic knowledge-intensive processes.
- *Lesson 2*: The current focus on business processes should not result in a blind spot for the 'product' being manufactured. The status of the product in terms of information objects should be visible and used as a starting point for any workflow design.

Based on these two lessons, we propose a paradigm shift and a tool to support product-driven case handling.

3. PRODUCT-DRIVEN CASE HANDLING

In this section we first introduce the concepts and then provide a meta model for *Product-Driven Case Handling* (PDCH).

3.1 PDCH Concepts

The central concept for PDCH is the *case* and not the activities or the routing from one worktray to another. The case is the 'product' which is manufactured and at any time workers should be aware of this context. Examples of cases are the evaluation of

a job application, the verdict on a traffic violation, the outcome of a tax assessment, and the ruling for an insurance claim.

To handle a case, *activities* need to be executed. Activities are logical units of work. Many WFM systems impose the so-called ACID properties on activities. This means that an activity is considered to be atomic and either carried out completely or not at all. We use a less rigid notion. Activities are simply chunks of work which are recognized by workers, e.g. like filling out an electronic form. As a rule-of-thumb, activities are separated by points where a transfer of work from one worker to another is likely/possible. Please note that activities separated by points of 'work transfer' can be non-atomic, e.g., the activity 'book business trip' may include tasks such as 'book flight', 'book hotel', etc.

Clearly activities are related and cases follow typical patterns. A *process* is the 'recipe' for handling cases of a given type. In many WFM systems, the specification of process fixes the routing of cases along activities and workers have hardly any insight in the whole. As a result exceptions are difficult to handle because they require unparalleled deviations of standard recipe. Since "exceptions are the rule", precedence relations among activities should be minimized.

If the workflow is not exclusively driven by precedence relations among activities and activities are not considered to be atomic, then another paradigm is needed to support the handling of cases. Workers will have more freedom but need to be aware of the whole case. Moreover, the case should be considered as a 'product' with structure and a current state. For knowledge-intensive processes, the state and structure of any case is based on a collection of *data objects*. A data object is a piece of information which is present or not present and when it is present it has a value. In contrast to existing WFM systems, the logistical state of the case is not determined by the control-flow status but by the presence of data objects. This is truly a paradigm shift: PDCH is also driven by data-flow instead of exclusively by control-flow. This provides a balance between the data-oriented approaches of the 80-ties and the process-oriented approaches of the 90-ties.

It is important that workers have insight in the whole case when they are executing activities. Therefore, all relevant data should be presented to the worker. Moreover, workers should be able to look at other data objects associated to the case they are working on (assuming proper authorization). *Forms* are used to present different views on the data objects associated to a given case. Activities can be linked to a form to present the data objects most relevant.

Forms are only a way of presenting data objects. The link between data objects, activities, and processes is specified directly. Each data object is linked to a process. So-called *free* data objects can be changed while the case is being handled. A data object that is explicitly linked to an activity is either *mandatory* or *restricted*. If a data object is mandatory, it is required to complete the activity. If a data object is restricted, then it is required to complete the activity and it cannot be entered in preceding or subsequent activities. That means that the information can be processed if and only if at least one of the activities for which the information is restricted is now at hand.

Actors are the workers executing activities and are grouped into *roles*. Roles are specific for processes, i.e., there can be multiple roles named 'manager' as long as they are linked to different processes. One actor can have multiple roles and roles may have multiple actors. Roles can be linked together through role graphs. A role graph specifies 'is_a' relations between roles. This way, one can specify that anybody with role 'manager' also has the role 'employee'.

For each process and each activity three roles need to be specified: the execute role, the redo role, and the skip role.

- The *execute role* is the role that is necessary to carry out the activity or to start a process.
- The *redo role* is necessary to undo activities, i.e., the case returns to the state before executing the activity. Note that it is only possible to undo an activity if all following activities are undone as well.
- The *skip role* is necessary to pass over activities. In order to skip over two consecutive activities, the worker needs to have the skip role for both activities.

The three types of roles associated to activities and processes provide a very powerful mechanism for modeling a wide range of exceptions. The redo ensures a very dynamic (as it is dependent on the role of the employee and the status of the case) and flexible form of a *loop*. The skip takes care of a range of exceptions that would otherwise have to be modeled in order to pass over activities. Of course, there are ways of avoiding undesirable effects: you can define the '*no-one'* or '*nobody'* role in every process that is higher than all the other roles and that no user can perform. You can also define an '*everyone*' role that is lower than all others. An activity with the '*no-one'* redo role can never be undone again and it would then also not be possible to go back to an earlier activity. This is a very effective way to model 'points of no return'. An execute *everyone* role means that the activity can be carried out by anyone who at least has a role in that process (because that person is then, after all, at least equal to the *everyone* role).

3.2 PDCH UML meta model

To structure the concepts introduced thus far, we use a UML object model. The object model shown in Figure 1 describes the concepts at a meta level.

The classes *case*, *process*, *activity*, *data object*, *data value*, *form*, *role*, and *actor* correspond to the entities discussed in this paper. Association *instance_of* links cases to processes. Each case is linked to one process but one process can be linked to many cases. Association *consists_of* shows that a process may contain many activities but each activity is part of a single process. Association *p_has_roles* shows that this also holds for roles. There is an m-to-n correspondence between actors and roles as is indicated by the association *a_has_roles*. Association *is_a* specifies the role graph. Forms are linked to activities and data objects. This is indicated by the associations *has_form* and *has_f_data*. Note that there may be forms which are not connected to any activity. These forms are linked directly to a process as expressed by the association *has_p_form*. This refers to the essence of case handling: values of data object may be viewed or updated (assuming proper authorization) without executing activities. Each form corresponds to one process. This is not shown in Figure 1 but can easily be added by an additional association between *form* and *process*.

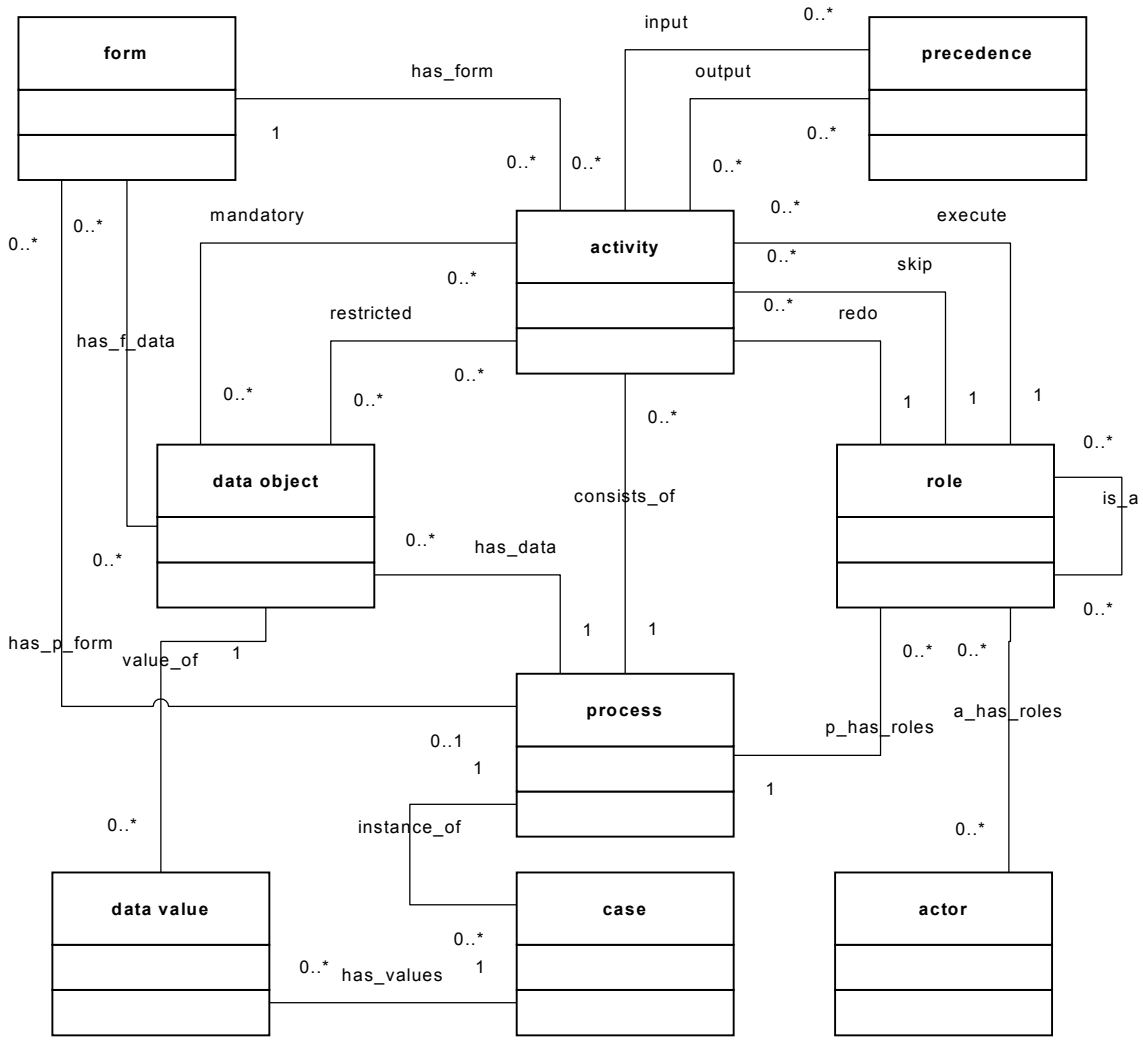


Figure 1: PDCH meta model.

The object class *precedence* contains precedence relations. Like in any WFM system these precedence relations are used to calculate which activities are first in line to be executed [1]. It is possible to use these relations to model sequential behavior. If a precedence relation has multiple outputs, then it can be used to model an AND-split or an OR-split. If a precedence relation has multiple inputs, then it can be used to model an AND-join or an OR-join. Using these precedence relations one can specify all routing constructs available in traditional WFM systems. However, the associations *mandatory*, *restricted*, *redo*, *skip*, and *execute* show that there are additional mechanisms to handle cases.

The associations *redo*, *skip*, and *execute* link roles to activities. Traditional WFM systems only support the execute role. The redo and skip roles allow for more flexibility. Note that a meaningful definition of these roles is only possible for acyclic workflow graphs, i.e., cyclic precedence relations are not allowed. Skip and redo actions require a notion of 'before' and 'after'. In cyclic workflow graphs, 'before' and 'after' are ill-defined. Note that iteration can be emulated using execute and redo. This is consistent with the statement "workflow loops are exceptions" raised by several authors [6]. The skip and redo roles lead to a very dynamic behavior: not just specific loops are modeled, but a whole range of loops, depending on the role(s) of the user working on the case.

The associations *mandatory* and *restricted* link data objects to activities. The association *has_data* denotes the link between data objects and processes. These data objects are either free and not linked to a specific activity or they are linked to specific activities through the associations *mandatory* and/or *restricted.* Mandatory data objects need to be present to complete the corresponding activity. Restricted data objects can only be entered when the corresponding activity is enabled. Although the associations *mandatory* and *restricted* suggest a tight coupling between data objects and activities, data objects are really linked to processes/cases and roles/actors. Free data objects can be changed and/or entered while executing the activity but also before and after executing the activity. Mandatory data objects can also be entered before, during, or after the execution of the activities linked through association *mandatory*. However, an activity can only be completed once all mandatory data objects are

present. It is important to note that a mandatory data object may appear in forms not linked to one of the corresponding activities. This means that mandatory data objects can be entered in advance. The value of a mandatory data object can only be entered or changed if the employee has a role that is equal to or higher than the execute role in at least one of the activities. An example of such a mandatory data object is the policy number of a client. The policy number may be entered before the value is really required for further processing. If a data object is restricted to one or more activities, the value of the data object can only be entered or changed if one of the activities is, in fact, the next one due to be carried out at that moment. An example is the 'Approved?' data object that can only be carried out if the activity 'approve insurance claim' is next in line.

3.3 PDCH dynamics

Figure 1 gives a static view of the case handling process. The state of a case is completely determined by objects of the class *data value*. Each object of this class is linked to a concrete case and a data object. This indicates whether the value of a data object is present and if it is present, the corresponding value is given. Note that in traditional WFM systems the state of a case is based on a 'control-flow pointer', e.g., if Petri nets are used, the configuration of tokens [1]. For case handling, the data objects unambiguously specify the state and enabled activities. To describe the corresponding mechanism, we first give the five potential states of an *activity instance*, i.e., an activity which may or may not be executed for a given case: (1) *initial*, (2) *enabled*, (3) *completed*, (4) *undone*, and (5) *skipped*. An activity instance starts in state *initial*. An instance becomes *enabled* if all preceding activities have been completed or skipped. An instance becomes *skipped* if the activity instance was enabled but skipped explicitly by a user with the proper skip role or if there was a choice in the process (i.e. an OR-split) resulting in a scenario not enabling the activity. An instance becomes *undone* if the activity instance was executed but was rolled back via a redo action. An activity instance is *completed* if and only if:

- all previous activities have been completed (or skipped),
- all mandatory/restricted data objects of an activity have a value, and
- the so-called *completion condition* of an activity is true.

The completion condition is normally set to 'TRUE' which means that it is sufficient to give all mandatory data objects a value. Note that the fact whether an activity is completed only depends on the values of data objects. This illustrates the changeover from control-flow to PDCH.

4. FLOW*er*

In this section, we give a brief description of FLOW*er*, Pallas Athena's case handling product. *FLOWer* can be used for flexibly structured processes, but also has the functionalities necessary for traditional production workflow. FLOW*er* compensates for many of the shortcomings of the traditional WFM systems and offers organizations new mechanisms to respond effectively to change. Flexibility is guaranteed through data-driven workflows, redo and skip capabilities, and activity independent forms. FLOW*er* goes beyond workflow: It fully supports the concepts appearing in the meta model shown in Figure 1.

4.1 FLOW*er* architecture

Figure 2 shows the architecture of FLOW*er*. FLOW*er* consists of a number of components: FLOW*er* Studio, FLOW*er* Case Guide, FLOW*er* CFM, FLOW*er* Integration Facility, and FLOW*er* Management Information and Case History Logging.

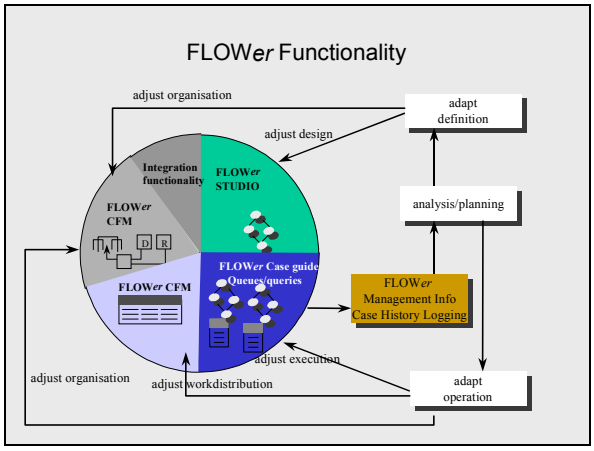


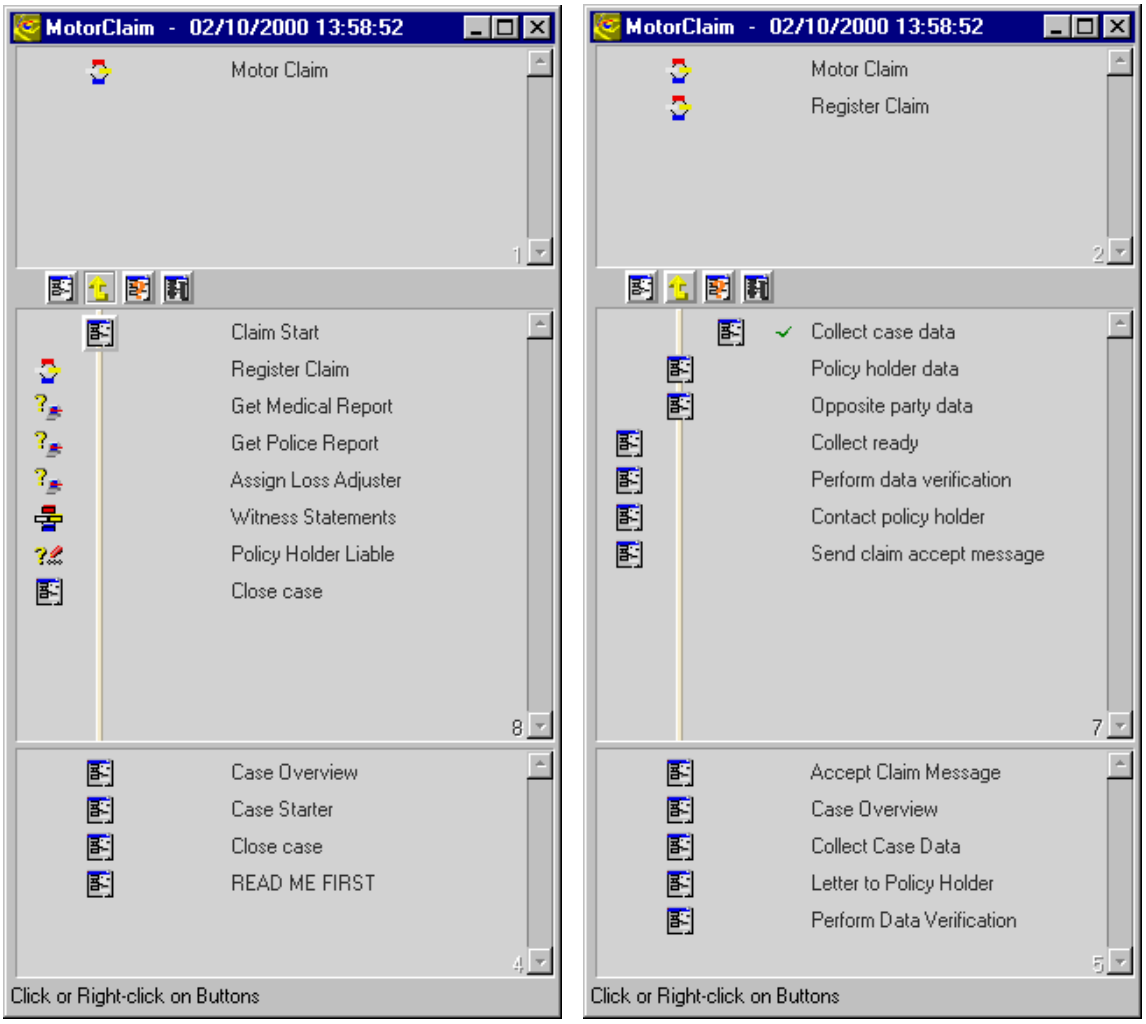
Figure 2: The architecture of FLOW*er*.

FLOW*er Studio* is the graphical design environment. It is used to define processes, activities, precedences, data objects, and forms. FLOW*er Case Guide* is the client application which is used to handle individual cases. FLOW*er queue* corresponds to the worktray, worklist or in-basket of traditional WFM systems. The FLOW*er* queue provides a refined mechanism to look for cases satisfying specified search criteria. FLOW*er CFM* (ConFiguration Management) is used to define users (i.e. actors), work profiles, and authorization profiles. The profiles are used to map users onto roles. FLOW*er* CFM is also available at the operational level to allow for run-time flexibility. FLOW*er Management Information and Case History Logging* can be used to store and retrieve management information at various levels of detail. FLOW*er Integration Facility* provides the functionality to interface with other applications. The following services are supported on the client side: COM objects (synchronous), Command Line Interface integration, DLL (NT/Windows) or shared libraries (UNIX), parameterized invocation of client applications such as MS Excel, MS Word etc. The following is available on the server side: C-API, SQL integration to access external databases, APPC (separately available C module), and DCOM (via C wrapper). The basic FLOW*er* functionality is also available through an API (Application Program Interface). This allows you, for example, to develop your own client or to use FLOW*er* as a workflow (or even better – a case handling) engine. In the remainder we focus on the end-user interface and the design interface of FLOW*er*. The first component (FLOW*er* Case Guide) illustrates the difference between WFM systems and case handling tools from a user perspective. The second component (FLOW*er* Studio) is used to discuss some of the more advanced features.

4.2 FLOW*er* Case Guide

The traditional worktray supported by the traditional WFM systems is used to push work items (i.e. activities enabled for a specific case) to workers without presenting the proper

context and reducing operational flexibility. FLOW*er* Case Guide, the client application of FLOW*er*, is very different. First of all, the whole case is shown, i.e., there is no context tunneling. Second, workers are not forced to execute activities in a fixed predefined order.



(a) (b)

Figure 3: Two screenshots of the case guide of FLOW*er*.

Figure 3 shows two screenshots of the case guide. The screenshots used in this section are taken from a process for dealing with (motor) insurance claims. The left screen, i.e., Figure 3(a), corresponds to the initial screen when initiating a new case. The case guide window consists of three zones. The top zone provides the context and can be used to navigate through the process. The zone below this one provides the contents of that sub-process with the status line and activities, sub-processes and decisions. To the left of the *status line* (also called the time line or wave front) are all the process objects that still need to be handled, on the status line is what is due to be processed now and to the right of the line is what has been completed. The objective is to get all the objects to the right of the status line. If an activity is colored (i.e. not white), the employee can carry it out. The bottom zone contains the forms. The user clicks on the first activity, *Claim Start*, which opens the first form. The first form is used to initialize the insurance claim. After executing activity *Claim Start* the corresponding icon is moved to the right side of the status line. Since *Register Claim* is specified as the next step in the process, it is moved on the status line. The icon corresponding to *Register Claim* indicates that it is not an activity but a sub-process. FLOW*er* allows for sub-processes. (This functionality will be discussed in more detail later.) *Witness Statement* is also a sub-process. As Figure 3(a) shows, activities corresponding to choices have different icons. The zone at the bottom of the window indicates that there are four forms directly linked to the process at this level. Note that these forms can be opened at any point in time, i.e., the status line only suggests the execution of activities on the status line. A user with proper authorization can open each of these forms at any time and can implicitly progress the state of the case through the addition of information.

Figure 3(b) shows the case guide after starting the sub-process *Register Claim* and successfully completing the first activity in this sub-process. The top zone gives context information: The window provides information about a case of the process *Motor Claim* with the locus of control confined to the sub-process *Register Claim*. The zone in the middle indicates that the first activity in *Register Claim* has been completed and that there are two activities on the status line: *Policy holder data* and *Opposite party data*. Both activities are enabled. Note that inside the sub-process different forms are offered to the worker.

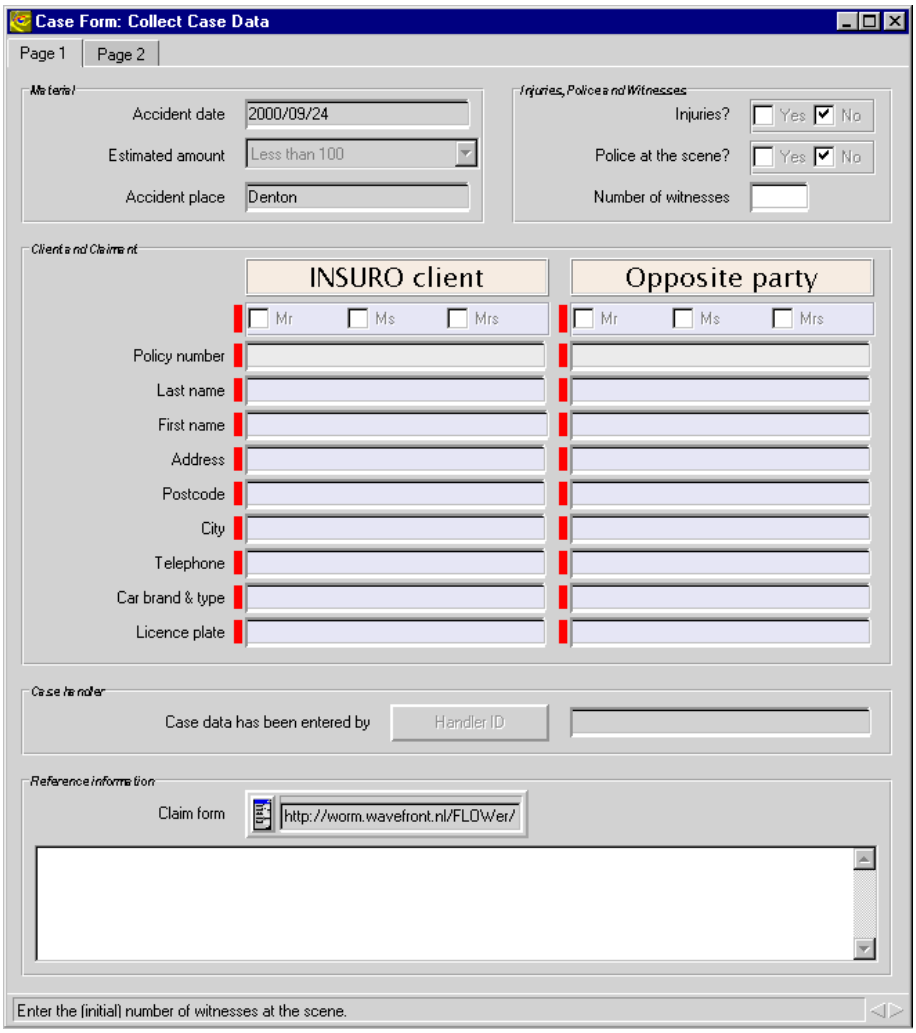


Figure 4: A FLOW*er* form containing free, mandatory, and restricted data objects.

Figure 4 shows a FLOW*er* form. This form corresponds to the activity *Collect case data*, i.e., the first activity of sub-process *Register Claim*. The form provides a view on a subset of the data objects associated to the case.

FLOW*er* forms have all the options that are standard for current form packages. A form can be linked to zero or more activities or sub-processes. On the other hand, only one form can be connected to an activity. The statuses of the data objects on a FLOW*er* form are shown to the user. FLOW*er* indicates, for example:

- those data objects that are mandatory in order to complete the activity that is now on the status line and to which this form is connected;
- those data objects that already have a value but that need to be re-confirmed or adapted because a previous activity was carried out again.

Because a form can be connected to more than one activity, the user can carry out various activities through one form and in a single session. If all mandatory data objects have been entered, FLOW*er* automatically completes the activity and immediately shows the mandatory fields belonging to the next activity that is connected to this form. The user therefore does not need to return to a work tray to start up the following activity. In other words, an experienced employee can process a case largely through the forms. Moreover, FLOW*er* also ensures that an employee can only enter information on those fields for which he or she has permission (on the basis of the Role Model and the status of the case). This allows you to model that an employee with a low role

can only fill in a part of the form, and an employee with a high role can fill in the entire form.

4.3 FLOW*er* Studio

The process design environment of FLOW*er* is called *Studio*. In this graphic design environment a process can be designed in full. A design consists broadly of four elements:

1. *Process flow* to determine the sequence of processing. The available object types are described below;
2. *Role graph* in which the roles and their interrelationships are recorded. A role represents an authorization level within a process. The roles and role graph are defined for each process;
3. *Data* that is important for handling the process;
4. *Forms* where the data can be shown and entered.

Let us first consider the process flow, i.e., the way precedence relations etc. are handled in FLOW*er*. The following types of process objects or *nodes* are available:

- Activity, with three sub-types, namely an activity connected to: a FLOW*er form*, a *standard letter*, or an *application*. A standard letter is a feature to automatically create documents - with data text integration - allowing all the data of the case to be used.
- Sub-plan or sub-process, also consisting of three types: *static*, *dynamic*, or *sequential*.
- Decision, in two types: *user decision* and *system decision*. A user decision is taken by the user during processing. A system decision is taken by FLOW*er* during processing, based on the data known at the time.

The process objects mentioned are used in the process flow. A process flow is, in fact, a flow diagram, or *graph*. The process flow describes the order in which the process objects should be handled. A sub-process is an object that is also a graph. In this way a tree structure is formed with a main process and below it, the sub-processes.

The UML meta model given in Figure 1 did not incorporate sub-processes. Since FLOW*er* supports three sub-processes relevant for case handling we discuss the three types of sub-processes in more detail:

- *Static sub-process*. A static sub-process represents a part of the process that is grouped for purposes of structuring. An example is 'Register Claim Data' in the case of a motor claim. This sub-process contains a number of activities, all of which relate to registering the claim.
- *Dynamic sub-process*. A dynamic sub-process represents a part of the process that needs to be repeated a number of times during processing. An example is the 'Process Witness' function in the case of a motor claim. This sub-process needs to be repeated for each witness. It goes without saying that at any time the employee could have more than one witness to deal with. Note that a dynamic sub-process is not the same as iteration. In the case of iteration, part of the process is *carried out again* whereby the previous result is overwritten. In a dynamic sub-process a number of separate sub-processes are created. In the example mentioned, there is exactly one sub-process for each witness.
- *Sequential sub-process*. A sequential sub-process is similar to a dynamic sub-process with the difference that, with sequential sub-processes, only one sub-process can ever be worked on at a time. In other words, a new *instance* is only possible after the previous one has been completed. An example is 'Request Information' in the case of a motor claim. If the information does not arrive in time, the first sub-process is terminated and a new *instance* is created, whereby the information is again requested. A sequential sub-process is particularly suitable for modeling reminders and diary entries.

The static sub-process is supported by most WFM systems. However, as indicated in [4], many workflow systems have problems dealing with sub-processes which are instantiated multiple times at run-time. The dynamic sub-process and the sequential sub-process constructs of FLOW*er* are powerful mechanisms to deal with these multiple instances inside a single case. Note that sub-processes were not considered in Figure 1.

FLOW*er* uses a *role* to indicate a permission or authorization level. A role is always connected to a process. This allows us to define the role of 'Senior Handler', for example, for the 'Settle Motor Claim' process and the role 'Senior Handler' for the 'Process Credit Application' process. These roles are different because a role exists within the context of a process. An employee has the role of 'Senior Handler', for example, for the 'Settle Motor Claim' but not for the 'Process Credit Application' process. Besides, it is also possible to connect an employee to a function to which a collection of roles (with their accompanying processes) can be connected. These possibilities mean optimum flexibility. Roles can also be used to screen a process. You can opt, for example, to model that only an employee with the 'Medical Adviser' role can view the part of a process (activities, data etc.,) that relates to medical matters.

Organizations are modeled by collections of roles for different processes. It is possible to define that one role is higher than the other role by placing the roles in a so-called role graph. A higher role can always do everything that a lower role can do, in other words there is succession, except for the screening functionality. In that case even higher roles are not allowed to even see, let alone carry out, the screened process parts. The relationships between the different roles can be specified graphically using a so-called role graph.

FLOW*er* uses the relationships in the role graph to determine what a user can do based on his/her roles. To do this, FLOW*er* uses the role model described in the meta model, i.e., the process designer has to define three roles for each process object (node): (1) the *execute role*, (2) the *redo role*, and (3) the *skip role*. The execute role is the role that is necessary to carry out an activity or to start a sub-process. This role is consistent with what the WfMC calls a role [15]. The WfMC does not identify the other two types (i.e., redo and skip). Nevertheless, they are essential for creating operational flexibility. The redo role on an activity is the role that is necessary to return the case to before that activity. Suppose that, in a case such as 'Settle Motor Claim', an employee wants to redo an activity carried out earlier, such as 'Register Claim Data'. This means that all the interim activities also need to be carried out again. The employee may only do this if he/she has a role that is at least as high as all the redo roles of the interim activities and the 'Register Claim Data' activity itself. Also he/she must then at least have the execute role of 'Register Claim Data' in order to handle this activity again. The skip role is necessary to pass over an activity. In order to skip two consecutive activities, for example, the employee must have a role that is at least equal to the skip role of those two activities.

In administrative processes, progress is determined by the data that is present. This is a generally accepted fact. Some of this

data is purely process control data that can be used, for example, in decisions. In this way the typing of a case as 'straightforward' or 'complex' determines the subsequent activities. Other data is purely content-related such as the telephone number of a client, for example. But some data includes both aspects and is relevant for both the content and the control. An example is the level of the claim amount that determines the control, but is also defined in a primary (or legacy) system.

FLOW*er* has extensive possibilities for defining and typing data, including *structures* and *arrays*. It is even possible to define *all* data (so including all content data) in FLOW*er*. But this is not necessary, of course. With the integration functionality of FLOW*er*, this type of data can also be stored only in the primary systems.

FLOW*er* is completely data driven and uses data to determine the state of a case. A data object within a process can be connected to zero or more activities. The nature of this connection determines the importance of the data object for the activity. FLOW*er* supports the types *free*, *mandatory,* and *restricted* mentioned earlier. An activity has been completed if all previous activities have been completed (or skipped), all mandatory/restricted data objects of an activity have a value, and the *completion condition* of an activity evaluates to true.

If an employee wants to give a restricted data object a value, the accompanying activity must be due to be carried out (be positioned on the status line) or the employee must move the status line by *skipping* the activities that are in between or by *redoing* the activity again. This is only possible if he or she has the appropriate role, as described in the FLOW*er* role model.

Moving the status line influences the activities that had already been carried out and are now in front of the status line again. These now have to be carried out again. In FLOW*er* it is not necessary to give all involved data objects a value again. FLOW*er* remembers the value entered earlier and gives the data objects a special status, 'awaiting confirmation'. The user can then confirm the value or values for each data object or activity.

The fact that FLOW*er* Studio allows for the definition of powerful links between activities, roles, and data objects results in an extremely flexible model where the user, depending on his or her role, can process a very large number of exceptions. But, on the other hand, by using restricted data objects and the 'no-one' role, it is also relatively simple to force the sequential order of handling.

A detailed discussion on FLOW*er* CFM, FLOW*er* Integration Facility, and FLOW*er* Management Information and Case History Logging is beyond the scope of this paper. Using these tools it is possible to fine-tune the operational and organizational aspects of case handling, to integrate with a variety of applications ranging from legacy systems to web-based solutions, and to obtain detailed management information. It is important to note that FLOW*er* clearly distinguishes *authorization* and *distribution* aspects. The process model does not contain any references to specific behavior at run-time. E.g. it is possible that two workgroups use different distribution mechanisms. This is clearly an characteristic of the work distribution and not of the process.

The interested reader is referred to an on-line demo which shows the functionality of FLOW*er*: http://www.pallas-athena.com/.

5. RELATED WORK

Many authors have addressed the issue of workflow flexibility. In recent years, there have been many workshops, edited books, and special issues of journals on workflow flexibility [3,5,13,14]. Agostini and De Michelis [6] argue that very simple workflow models should be used and exceptions should be dealt with by hand through so-called "linear jumps'. Other authors, e.g., [7], give concrete adaptation rules. Some authors even state that "workflow change is a workflow" [8]. Several authors propose a more declarative style of specifying workflows. Consider for example the Vortex paradigm [11]. These are just a few pointers to the elaborate literature on workflow flexibility.

The problems with respect to designing process models for real-life processes have been recognized in [2,10,19]. Herrmann [10] seeks a solution by using semi-structured workflow models. Reijers et al. [2,19] propose the product-driven approach adopted in this paper.

Vendors of WFM systems have also been struggling with flexibility issues. Systems such as InConcert (TIBCO) allow for ad-hoc routing of workflow instances (i.e. cases). However, these systems require on-the-fly modifications of process models by end-users. Vectus (London-Bridge/Hatton Blue) is one of the few systems also aiming at case handling. Compared to FLOW*er*, the focus is more on Customer Relationship Management (CRM) than supporting a variety of workflow processes. Recently, Staffware extended their workflow management system with a case handler. Unfortunately, the case handler only works within the context of a single step in the process.

6. CONCLUSION

There are currently very few, if any, workflow systems available that can be applied to flexible processes with frequent exceptions and where individual employees have to be able to carry out a wide range of different steps. This type of functionality is called 'operational flexibility'. Most standard workflow packages do not support this because each exception needs to be explicitly modeled. Furthermore, there is often no functionality available for allowing one user to control several steps in a process. Standard workflow packages also fail to provide another type of process support: case handling. Case handling involves approaching a case-related or folder-related task from the context of the entire folder. Those carrying out the process must have at all times full insight into all the tasks and activities that could be carried out or have already been carried out at any given moment.

To address these issues we have developed a meta model to support *product-driven case handling* (PDCH). The meta model has been used to identify the differences with conventional workflow management. It should be noted that contemporary WFM systems only support a specific form of case handling, i.e., case handling with context tunneling, rigid routing (i.e., no redo or skip and all data object restricted), and a push-oriented distribution of work items.

In this paper we also presented a system to support PDCH. FLOW*er* is unique in providing both operational flexibility as well as case handling facilities. The development of the first fully operational version of FLOW*er* was completed in the middle of 2000. Therefore, the installed base is limited. However, the diversity of the processes currently supported by FLOW*er* is impressive. Examples of case-handling processes currently managed by FLOW*er* are: application for Dutch citizenship, handling of house tax complaints, handling of complex social

benefits applications and complaints, handling of insurance claims, and handling of lawsuits.

7. ACKNOWLEDGMENTS

We thank all the people involved in the development of FLOW*er*. Special thanks go the FLOW*er* development team of Pallas Athena. We also thank the anonymous referees for their comments.

8. 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. On the automatic generation of workflow processes based on product structures. Computers in Industry, 39:97-111, 1999.

3 W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors. Business Process Management: Models, Techniques, and Empirical Studies, volume 1806 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2000.

4 W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, 7th International Conference on Cooperative Information Systems (CoopIS 2000), volume 1901 of Lecture Notes in Computer Science, pages 18-29. Springer-Verlag, Berlin, 2000.

5 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.

6 A. Agostini and G. De Michelis. Improving Flexibility of Workflow management Systems. In [3], pages 218-234.

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

8 C.A. Ellis and K. Keddara. A Workflow Change Is a Workflow. In [3], pages 218-234.

9 M. Hammer and J. Champy. Reengineering the corporation. Nicolas Brealey Publishing, London, 1993.

10 T. Herrmann, M. Hoffmann, K.U. Loser, and K. Moysich. Semistructured models are surprisingly useful for user-centered design. In Coop2000, Sophia-Antipolis, France, May 2000.

11 R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative Workflows that Support Easy Modification and Dynamic Browsing. In Proceedings of International Joint Conference on Work Activities Coordination and Collaboration (WACC'99), pages 69-78, 1999.

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

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

14 M. Klein, C. Dellarocas, and A. Bernstein, editors. Adaptive Workflow Systems, special issue of the journal of Computer Supported Cooperative Work, volume 9, numbers 3-4, 2000.

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

16 A. Orlicky. Structuring the bill of materials for mrp. Production and Inventory Management, pages 19-42, Dec 1972.

17 E.A.H. Platier. A logistical view on business processes: BPR and WFM concepts (in Dutch). PhD thesis, Eindhoven University of Technology, Eindhoven, 1996.

18 G. Poyssick and S. Hannaford. Workflow Reengineering. Adobe Press, Mountain View, CA, 1996.

19 H. Reijers and K. Voorhoeve. On the Optimal Design of Processes and Information Systems (in Dutch). Informatie, 42:50-57, December, 2000.

20 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.