Available online at www.sciencedirect.com
SCIENCE DIRECT®
Data & Knowledge Engineering 53 (2005) 129–162
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Case handling: a new paradigm for business process support

Wil M.P. van der Aalst [a,*], Mathias Weske [b], Dolf Grünbauer [c]

[a] *Department of Technology Management, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands*
[b] *Hasso Plattner Institute for Software Systems Engineering, Prof.-Dr.-Helmertstrasse 2-3, 14482 Potsdam, Germany*
[c] *Pallas Athena, P.O. Box 747, NL-7300 AS, Apeldoorn, The Netherlands*

Available online 21 August 2004

Abstract

Case handling is a new paradigm for supporting flexible and knowledge intensive business processes. It is strongly based on data as the typical product of these processes. Unlike workflow management, which uses predefined process control structures to determine what should be done during a workflow process, case handling focuses on what *can* be done to achieve a business goal. In case handling, the knowledge worker in charge of a particular case actively decides on how the goal of that case is reached, and the role of a case handling system is assisting rather than guiding her in doing so. In this paper, case handling is introduced as a new paradigm for supporting flexible business processes. It is motivated by comparing it to workflow management as the traditional way to support business processes. The main entities of case handling systems are identified and classified in a meta model. Finally, the basic functionality and usage of a case handling system is illustrated by an example.

Keywords: Case handling; Workflow management systems; Adaptive workflow; Flexibility; Business process management

* Corresponding author. Tel.: +31 40 247 4295; fax: +31 40 243 2612.
E-mail addresses: w.m.p.v.d.aalst@tm.tue.nl (W.M.P. van der Aalst), weske@hpi.uni-potsdam.de (M. Weske), dolf.grunbauer@pallas-athena.com (D. Grünbauer).

0169-023X/$ - see front matter
doi:10.1016/j.datak.2004.07.003

1. Introduction

1.1. Context

During the last decade workflow management concepts and technology [6,7,21,26,31,32,35] have been applied in many enterprise information systems. Workflow management systems such as Staffware, IBM MQSeries Workflow, COSA, etc. offer generic modeling and enactment capabilities for structured business processes. By making graphical process definitions, i.e., models describing the life-cycle of a typical case or workflow instance in isolation, one can configure these systems to support business processes. Recently, besides pure workflow management systems many other software systems have adopted workflow technology, for example ERP (enterprise resource planning) systems such as SAP, PeopleSoft, Baan, Oracle, as well as CRM (customer relationship management) software.

However, there appears to be a severe gap between the promise of workflow technology and what systems really offer. As indicated by many authors, workflow management systems are too restrictive and have problems dealing with change [6,9,11,15,19,24,29,30,52]. In particular, many workshops and special issues of journals have been devoted to techniques to make workflow management more flexible [6,9,29,30]. Some authors stress the fact that models should be as simple as possible to allow for maximum flexibility [11]. Other authors propose advanced techniques to support workflow evolution and the migration of cases of one workflow model to another [15,52]. 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. In contemporary workflow technology, the only way to handle changes is to go behind the system's back. If users are forced to bypass the workflow system quite frequently, the system is more a liability than an asset. If the process model attempts to capture all possible exceptions [46], the resulting model becomes too complex to manage and maintain. These and many other problems show that it is difficult to offer flexibility without losing control.

1.2. Terminology

To illustrate the deficiencies of contemporary workflow management and to motivate the case handling paradigm, we use the metaphor of a blind surgeon. Before doing so we first introduce some standard workflow terminology. Workflow management systems are *case-driven*, i.e., they focus on a single process instance. [1] This means that only business processes describing the handling of one workflow instance in isolation are supported. Many cases can be handled in parallel. However, from the viewpoint of the workflow management system these cases are logically independent. To handle each case, the workflow management system uses the corresponding *workflow process definition*. The process definition describes the routing of the case by specifying the ordering of *activities*. Activities are the logical units of work and correspond to atomic pieces of work, i.e., each activity is executed by one worker (or another type of resource) and the result is either "commit work" or "abort and roll back".

[1] Please do not confuse "case-driven" processes with "case handling". The case handling paradigm can be used to support case-driven processes. However, conventional workflow technology can also be used to case-driven processes.

To specify the ordering of activities typically some graphical language such as Petri nets [1] or workflow graphs [52] is used. These languages allow for sequential, conditional, and parallel routing of cases. Some of the workflow management systems allow for more advanced constructs [8]. Typically, an activity which is enabled for a given case may be executed by many workers, and many workers may execute a given activity. To support the distribution of work, the concept of a *role* is used. A worker can have multiple roles, but an activity has only one role. If activity A has role R, then only workers with role R are allowed to execute activities of type A. Based on this information, the workflow management system works as follows: The corresponding workflow process definition is instantiated for each new case, i.e., for each case (e.g., request for information, insurance claim, customs declaration, etc.) a new workflow instance is created. Based on the corresponding workflow process definition, the workflow engine calculates which activities are enabled for this case. For each enabled activity, one work-item is put in the in-tray of each worker having the appropriate role. Workers can pick work-items from their in-tray. By selecting a work-item the worker can start executing the corresponding activity, etc. Note that, although a work-item can appear in the in-tray of many workers, only one worker will execute the corresponding activity. When a work-item is selected, the workflow management system launches the corresponding application and monitors the result of executing the corresponding activity. Note that the worker only sees work-items in his/her in-tray, and when selecting a work-item only the information relevant for executing the corresponding activity is shown.

1.3. Four problems

In this paper, we argue that the lack of flexibility and—as a result—the lack of usability of contemporary workflow management systems to a large extent stems from the fact that routing is the only mechanism driving the case, i.e., work is moved from one in-tray to another based on pre-specified causal relationships between activities. This fundamental property of the workflow approach causes the following problems:

- Work needs to be *straight-jacketed into activities*. Although activities are considered to be atomic by the workflow 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, workers can see all the work they are authorized to do. Moreover, a worker is not authorized to do anything beyond the work-items in her in-tray. Clearly, work distribution and authorization should not coincide. For example, a group leader may be authorized to do the work offered to any of the group members, but this should not imply that all this work is put in his worklist. Since distribution and authorization typically coincide in contemporary workflow management systems, 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.

It is worth noting that not only traditional workflow technology suffers from these problems. Recent approaches to flexible workflow management are still based on routing as the only mechanism for process support and, hence, suffer from the problems mentioned.

1.4. Blind surgeon metaphor

We use the "blind surgeon metaphor" to illustrate the four problems identified by placing them in a hospital environment. In a hospital both operational flexibility and well-defined procedures are needed. Therefore, workflow processes in a hospital serve as benchmark examples for flexible workflow management, cf. [39]. Note that the "blind surgeon metaphor" is not restricted to hospital environments, similar issues can be observed in a wide range of other knowledge-intensive application scenarios.

Consider the flow of patients in a hospital as a workflow process. One can consider the admission of a patient to the hospital as the creation of a new case. The basic workflow process of any hospital is to handle these cases. The activities in such a workflow include all kinds of treatments, operations, diagnostic tests, etc. The workers are, among others, surgeons, specialists, physicians, laboratory personnel, nurses. Each of these workers has one or more roles, and each task requires a worker having a specific role. For example, in case of appendicitis the activity "remove appendix" requires the role "surgeon". Clearly, we can define hospital workflows in terms of process definitions, activities, roles, and workers.

In the setting of "hospital workflows", we again consider the four problems identified before. Suppose that work in hospitals would be *straight-jacketed into activities*. This would mean that workers would only execute the actions that are specified for the activity, i.e., additional actions would not be allowed, and it would also not be possible to skip actions. Such a rigorous execution of the work specified could lead to life-threatening situations. In hospital environments it is crucial that knowledgeable persons can decide on activities to perform based on the current case and their personal experiences. In general, workflow process models cannot represent the complete knowledge of the experts and all situations that might occur.

Suppose that the routing in hospital processes would be used for both *work distribution and authorization*. This would mean that activities can only be executed if they are in the in-tray of a worker. Since distribution and authorization then coincide, it would not be possible to allow for initiatives of workers, e.g., a physician cannot request a blood test if the medical protocol does not specify such a test.

Context tunneling is also intolerable. This would mean that the information for surgeons, specialists, physicians, laboratory personnel, and nurses is restricted to the information that is needed for executing a specific task. In contrast, given a specific medical situation, doctors and nurses may take advantage from consulting the complete medical record of the patient, based on the current state of the patient and their personal knowledge and experiences.

Finally, it is clearly undesirable that the medical staff of a hospital would limit their activities to what *should* be done according to the procedure rather than what *can* be done. The medical protocol typically specifies what should be done instead of what can be done. Such descriptions are useful to guide workers. However, it is clear that restricting the workers to the workflow specified in the medical protocol would lead to absurd situations.

It is clear that such a ''tunnel vision'', i.e., a straight-ahead vision without attention for contextual information, is not acceptable in any hospital process. Consider for example a surgeon who would ignore all information which is not directly related to the surgical procedure. A straightforward implementation of such a process using contemporary workflow management systems would result in surgeons who are blind for this information, just doing the actions specified for the activities in their in-trays. This ''blind surgeon metaphor'' illustrates some of the key problems of present-day workflow management technology.

1.5. Case handling

In this paper, we propose *case handling as a new paradigm for supporting knowledge-intensive business processes*. By avoiding the blind surgeon metaphor, a wide range of application scenarios for which contemporary workflow technology fails to offer an adequate solution will benefit from this new paradigm. The core features of case handling are:

- avoid context tunneling by providing all information available (i.e., present the case as a whole rather than showing just bits and pieces),
- decide which activities are enabled on the basis of the information available rather than the activities already executed,
- separate work distribution from authorization and allow for additional types of roles, not just the execute role,
- allow workers to view and add/modify data before or after the corresponding activities have been executed (e.g., information can be registered the moment it becomes available).

Based on these key properties, we believe that case handling provides a good balance between the data-centered approaches of the 80-ties and the process-centered approaches of the 90-ties. Inspired by Business Process Re-engineering (BPR) principles [22] workflow engineers have focused on processes neglected the products being produced by these processes [2]. Case handling treats both data and processes as first-class citizens. This balance seems to be highly relevant for knowledge intensive business processes.

This paper builds on the results presented in [5], where we focused on case handling in the context of a specific case handling tool named FLOW*er* [13]. Besides FLOW*er* of Pallas Athena there are few other case handling tools. Related products are ECHO (Electronic Case Handling for Offices), a predecessor of FLOW*er*, the Staffware Case Handler [44] and the COSA Activity Manager [43], both based on the generic solution of BPi [14], and Vectus [33,34]. Instead of focusing on a specific product, we generalize some of the ideas used in these tools into a conceptual model which clearly shows the difference between case handling and traditional workflow management. Then, we demonstrate the applicability of the case handling concept using FLOW*er*.

1.6. Outline

The remainder of this paper is organized as follows. Section 2 introduces case handling by focusing on the differences between case handling and traditional workflow management. Section 3 presents a conceptual model which describes the key features of case handling. Case handling

environments are precisely characterized in Section 4 by a mathematical formalization of their static and dynamic aspects. Note that Sections 2–4 are tool independent. Section 5 describes the case handling system FLOW*er* using a realistic example. Then we provide pointers to current case handling applications based on FLOW*er*. Finally, we discuss related work and conclude the paper. In the conclusion we position case handling in a broader spectrum involving other approaches such traditional production workflow, ad hoc workflow, and groupware.

2. The case handling paradigm

The central concept for case handling is the *case* and not the activities or the routing. 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 workflow management systems impose the so-called ACID properties on activities [1,26]. This means that an activity is considered to be atomic and either carried out completely or not at all. Case handling uses 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 or 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 [8]. A *process* is the recipe for handling cases of a given type. In many workflow management systems, the specification of a 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 from the standard recipe.

Since in dynamic application environments 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 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 workflow management 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: case handling is also driven by data-flow instead of exclusively by control-flow.

It is important that workers have insight in the whole case when they are executing activities. Therefore, all relevant information 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 most relevant data objects. 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. All other data objects are explicitly linked to one or more activities as a *mandatory* and/or a *restricted* data object. If a data object is mandatory for an activity, it is required to be entered in order to complete the corresponding activity. If a data object is restricted for an activity, then it can only be entered in this activity or some other activity for which the data object is restricted. If data object D is mandatory for activity A, A can only be completed if D has been entered. If D is restricted to A and no other activities, D can only be entered in A. Note that D may be mandatory for activity A *and* restricted to A, i.e., mandatory and restricted are two orthogonal notions. Moreover, forms are independent of these two notions. For example, the form attached to an activity may or may not show mandatory/restricted data objects. However, if D is mandatory for activity A and restricted to only A, but not in the form linked to A, then this will cause a deadlock since it is not possible to complete A. Therefore, mandatory and/or restricted data objects are typically in the corresponding form. Moreover, in many cases the form will contain additional data elements which are either free or mandatory for other activities in the process.

Note that mandatory data objects can he considered as some kind of *postcondition*. This observation raises the question why there is not a *precondition* (i.e., data objects have to exist before execution) in addition or instead of this postcondition. This functionality can be obtained by adding a dummy activity just before the activity which requires a precondition, i.e., the dummy activity has a postcondition which can be interpreted as a precondition of the subsequent activity. In other words, the dummy acts as a guard.

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 types of 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 that is higher than all the other roles, i.e., no user has this role, and therefore, the corresponding action is blocked. 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'. Using "everyone" as an execute 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). Note that in addition to these three roles, one could consider additional roles, e.g., the "responsible role" or the "supervisor role". For a case one could also define the "case manager role", etc.

The variety of roles associated to a case or an activity shows that in case handling it is possible to separate authorization from work distribution. When using the classical in-tray, one can only see the work-items which need to be executed. The only way to get to a case is through work-items in the in-tray, i.e., authorization and work distribution coincide. For case handling the in-tray is replaced by a flexible *query mechanism*. This mechanism allows a worker to navigate through all active and also to completed cases. The query "Select all cases for which there is an activity enabled which has an execute role R" can be used to emulate the traditional in-tray. In fact, this query corresponds precisely to the work queue concept used in the in-tray of the workflow management system Staffware. By extending the query to all roles a specific worker can fulfill, it is possible to create a list of all cases for which the worker can execute activities at a given point in time. However, it is also possible to have queries such as "Select all cases that worker W worked on in the last two months" and "Select all cases with amount exceeding 80k Euro for which activity A is enabled". By using the query mechanism workers can get a handle to cases that require attention. Note that authorization is separated from work distribution. Roles are used to specify authorization. Standard queries can be used to distribute work. However, the query mechanism can also be used to formulate ad hoc queries which transcend the classical in-tray.

To conclude this section, we summarize the main differences between workflow management, as supported by contemporary workflow technology, and case handling (cf. Table 1). The focus of case handling is on the whole case, i.e., there is no context tunneling by limiting the view to single work-items. The primary driver to determine which activities are enabled is the state of the case (i.e., the case data) and not control-flow related information such as the activities that have been executed. The basic assumption driving most workflow management systems is a strict separation between data and process. Only the control data is managed. The strict separation between case data and process control simplifies things but also creates integration problems. For case handling the logistical state of a case (i.e., which activities are enabled) is derived from the data objects present, therefore data and process cannot be separated! Unlike workflow management, case handling allows for a separation of authorization and distribution. Moreover, it is possible to

Table 1
Differences between workflow management and case handling

	Workflow management	Case handling
Focus	Work-item	Whole case
Primary driver	Control flow	Case data
Separation of case data and process control	Yes	No
Separation of authorization and distribution	No	Yes
Types of roles associated with tasks	Execute	Execute, Skip, Redo

distinguish various types of roles, i.e., the mapping of activities to workers is not limited to the execute role.

3. The case handling meta model

After motivating case handling and introducing the basic concepts of this new paradigm in Sections 1 and 2, we now identify the main entities of case handling environments as well as their relationships. In doing that we move from a rather informal discussion towards more precise modeling of case handling environments. An object-oriented approach is used for this endeavor, since it provides powerful modeling constructs which proved to be adequate for dealing with the complexity in case handling. We use the de facto standard in object oriented analysis and design, the unified modeling language (UML); mainly its structural features are used. The *case handling meta model* represents artifacts which are required to define cases and environments in which cases are executed; it is shown in Fig. 1.

Case definition is the central class of the case handling meta model. Case definitions are either complex (cases with internal structure) or atomic (cases without internal structure), referred to as complex case definitions and activity definitions, respectively. Complex case definitions consist of a set of case definitions, resulting in a hierarchical structuring of cases in sub-cases and activities. In the case handling meta model, this property is represented by a recursive association between complex case definition and case definition. Obviously each complex case definition consists of at least one case definition, and each case definition may occur in at most one complex case definition, as represented by the cardinalities of that association in Fig. 1.

Since case handling is a data-driven approach, activity definitions are associated with data object definitions. In particular, each activity definition is associated with at least one data object definition. This association is partitioned into two main types, i.e., mandatory and restricted. If a data object definition is mandatory for an activity definition then the respective data value has to be entered before that activity can be completed; however, it may also be entered in an earlier activity. A restricted association indicates that a data value can only be entered during a particular activity.

Restricted and mandatory associations between activities and data are an important implementation vehicle for business process support, since an activity can only be completed if and when values for the mandatory data objects are provided. Activity definitions are also associated with forms definitions. Forms are used to visualize data objects which are offered to the user. Forms are closely associated with activities, and they are an important means to business process support. The fields displayed in a form associated with an activity correspond to mandatory as well as restricted data objects for that activity. [2] In addition, the definition of forms may also contain data objects that are mandatory for subsequent activities. This feature allows flexible execution of business processes, since data values can be entered at an early stage, if the knowledge worker decides to do so. Data object definitions may also be free; free data objects are not associated with particular activities; rather they are defined in the context of complex case definitions. Hence, they

[2] As indicated before, the form may not contain all mandatory/restricted data objects. However, this may cause deadlocks or other anomalies.

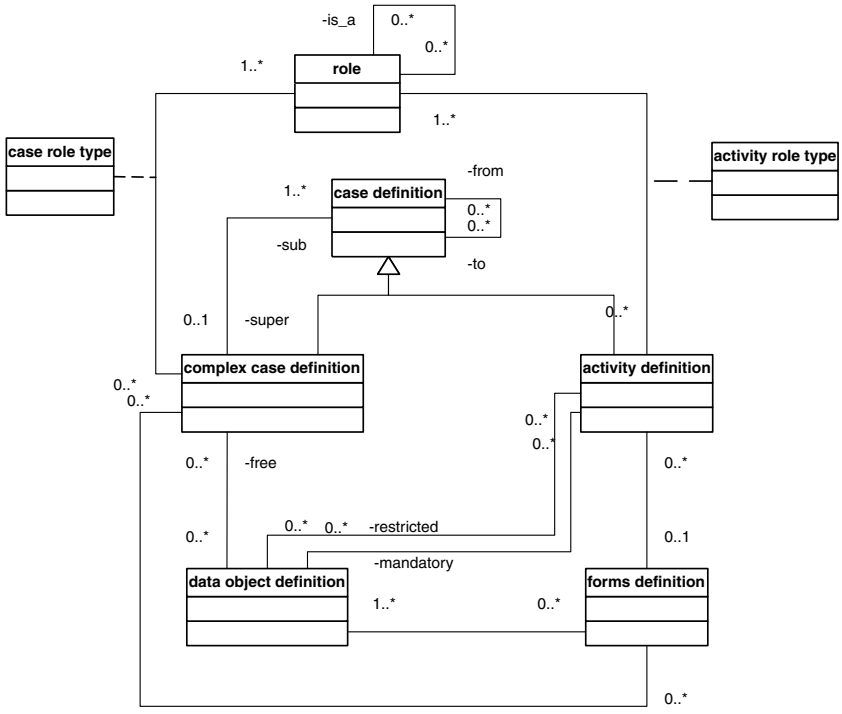

Fig. 1. Case handling meta model, schema level.

can be accessed at any time during the case execution. Free data objects are represented by an association of data object definition with complex case definition. The context of a case can be presented by such a form. As indicated above, providing the knowledge with as much information as possible is an important aspect of case handling systems.

Roles are used more thoroughly in case handling than in workflow management. In particular, there are multiple roles associated with a given case definition, and these roles have different types. Typical roles types associated with an activity are execute (to execute an activity), skip (to skip an activity that is not required during a particular case), and redo (to jump back to previous activities of the case with the option of re-doing these activities or re-confirming data object values which have already been entered). Role types associated with complex case definitions are, for example, manager and supervisor, to indicate persons which may manage or supervise complex cases; typically these roles are mapped to management personnel of an organization. Role types for activities are represented by an association class called *activity role type*, linking the role class and the activity definition class, while role types for complex cases are represented by an association class between the complex case definition and the role class.

The example shown in Fig. 2 illustrates the concepts introduced in the case handling meta model. It shows how cases, data objects and forms and their associations as well as organizational aspects are represented. We start by discussing the overall structure of the case definition. There is one complex case definition $C1$, which consists of activity definitions $A1$, $A2$, and $A3$, represented by the indirect recursion of complex case definitions and case definitions in the meta model, shown as a dotted line connecting $C1$ to its sub-cases. As shown in that figure, data object definition $D1$ is

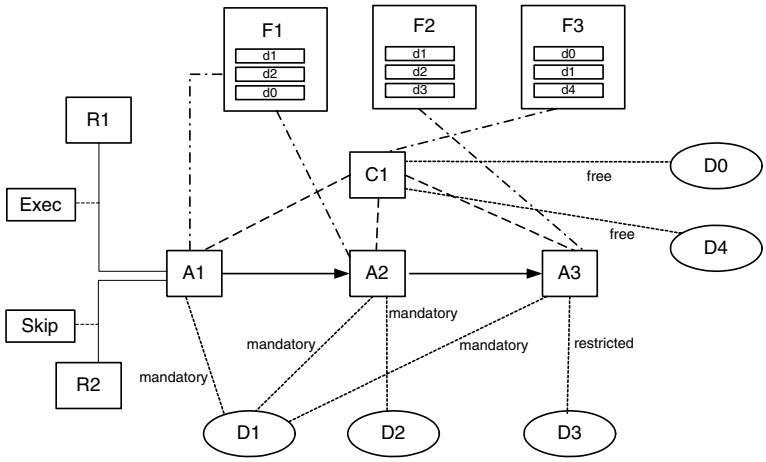

Fig. 2. Abstract example introducing the schema level of the case handling meta model.

mandatory for $A1$, $A2$ and $A3$. $D2$ is mandatory for $A2$, and $D3$ is restricted for $A3$. Since $D1$ is mandatory for $A1$, the form definition $F1$ associated with $A1$ holds a field for $D1$. However, there is also a field for $D2$ in that form. The knowledge worker in charge of a case based on that case definition may enter a value for $D1$ when $A1$ is ready for execution. In addition, she may also enter a value for $D2$ at this instant, which implicitly performs $A2$ as well. This is due to the fact that $D2$ is the only mandatory data object for $A2$. Notice, however, that $D3$ cannot be entered neither during $A1$ nor during $A2$, since it is restricted to $A3$ and can therefore only be executed in the context of $A3$, using the form associated with it.

The activities of the case are ordered: $A1$ is followed by $A2$ and $A3$, represented by the recursive association with roles *to* and *from* in the meta model. There are five data object definitions $D0$–$D4$. Dotted lines marked with association type names represent the associations between activity definitions and data object definitions. As indicated above, $D1$ is mandatory for $A1$, $A2$ and $A3$, $D2$ is mandatory for $A2$, while $D3$ is restricted for $A3$. $D0$ and $D4$ are free data elements, which appear in form definition $F3$, associated with the overall case definition $C1$. Notice that form definition $F1$ contains not only a field $d1$ representing data object definition $D1$ (mandatory for the completion of $A1$), but also $d2$ (for data object definition $D2$ which is mandatory for $A2$) and $d0$ (for data object definition $D0$ which is free). As discussed above, during the execution of $A1$ the knowledge worker may already enter a data value for $d2$, although this is not required for the completion of $A1$. However, $A1$ cannot complete before $d1$ is entered ($D1$ is mandatory for $A1$). The knowledge worker may use the information presented in $d0$ to work efficiently on the case. Not to overload the figure, the roles are not specified completely. In fact, only the roles for $A1$ are specified: $R1$ and $R2$ are associated with $A1$, where the association with $R1$ is of type execute (persons with role $R1$ may execute this activity), while the association with $R2$ is of type skip (persons with role $R2$ may skip this activity). This means that during the enactment of cases based on case definition $C1$, only knowledge workers which can play role $R1$ are permitted to perform activities based on $A1$, and only persons with role $R2$ may skip that activity.

Fig. 1 only shows entities at the schema level, i.e., entities such as (complex) case definitions, roles, activity definitions, data object definitions, and forms definitions. These entities are specified

at design-time. At run-time, other entities come into play, e.g., concrete cases, actors, activities, data objects, and forms. For example, a case definition "insurance claim" describes an insurance claim at the type level and not at the instance level. Case "insurance claim 993567 filed by Jones on August 10th" is an instantiation of case definition "insurance claim" and is example of an entity created and handled at run-time. Entities on the instance level are represented by the case handling model shown in Fig. 3. In this model concrete cases are in the center of attention. The overall structure of the object model shown in Fig. 3 is similar to the structure of the meta model shown in Fig. 1. For example, as case definitions are generalizations of complex case definitions and activity definitions in the meta model, cases are generalizations of complex cases and activities in the case handling model. Furthermore, there is a precedence ordering between cases, represented by a recursive relationship with roles *to* and *from* in both levels of abstraction. The main differences between the two models are the organizational embedding and the forms. In particular, while *role* is a class in the meta model, *actor* is a class in the case handling model. The cardinality of forms and form definitions are different in both models. In the meta model (schema level), each forms definition is associated with an arbitrary number of activity definitions, while in the case handling model (instance level) each form is associated with at most one activity. This is due to the fact that forms are instantiated for each activity with which they are associated. There are activities without forms to cater for automatic activities, for example automated queries to external database systems.

Fig. 3 assumes that at run-time the same form can be instantiated multiple times, i.e., if two activities share the same forms definition, there may be two copies of the same form. An alternative interpretation used by e.g. FLOW*er* is to see a form as simply a view on the data and not

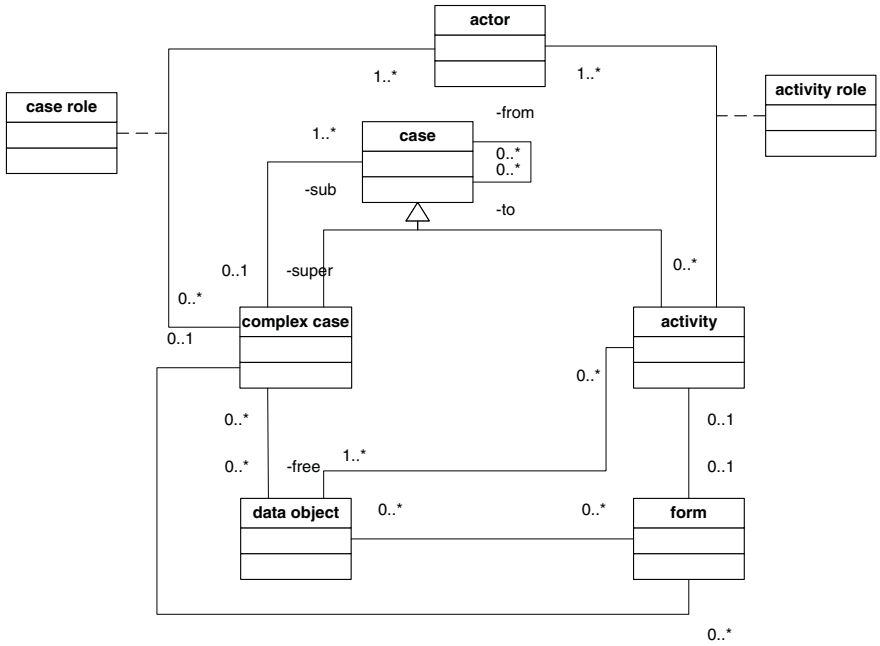

Fig. 3. Case handling meta model, instance level.

allow multiple instances of the same form for the same case at the same time. For this interpretation, the cardinalities in Fig. 3 should be like in Fig. 1.

4. A formal framework for case handling

This section formalizes most of the concepts introduced in the first half of this paper. The main purpose of this endeavor is to precisely describe the dynamics of a case handling environment, i.e., an execution model for case handling. Note that the meta model introduced in the previous section only considers static aspects. The meta model structures relevant entities at both the schema level and instance level. However, it does not specify the dynamics.

In this section, we will specify the dynamics using a formal model. First, we introduce a formal model describing a case definition. In this model, we abstract from certain entities (e.g., forms) and focus on activities and data objects. Based on this formal model, we describe the execution model for case handling in terms of state-transition diagrams and ECA-rules. Finally, we discuss the relation between the formal model and the entities excluded from the formal model, e.g., forms and actors.

4.1. Case definition

A case definition describes the way a case of a specific type is handled. Clearly, the case definition is a good starting point for formalizing the dynamics of case handling. For presentation purposes, we will limit our formalization of case handling to activities, data objects, and their interrelationships. These are the core entities which determine the execution semantics of case handling. The formalization will exclude forms and roles. Moreover, we do not consider nested case definitions, i.e., we assume that a case definition only contains activity definitions and not complex case definitions. Note that the latter is not a real limitation: Any hierarchical model can be flattened by recursively replacing complex case definitions by their decompositions. Forms and roles can be excluded because they only indirectly affect the execution semantics. Given these restrictions, we can define a case definition as follows.

Definition 4.1. A tuple $CD = (A, P, D, dom, mandatory, restricted, free, condition)$ is called *case definition*, if the following holds:

- A is a set of *activities definitions*,
- $P \subseteq A \times A$ is a *precedence relation*,
- D is a set of *data object definitions*,
- $dom \in D \mapsto 2^U$ is a function mapping each data object onto its *domain* (2^U denotes the power set of U), i.e., the domain of a data object definition is a set of values over some universe U,
- $mandatory \subseteq A \times D$ is a relation which specifies *mandatory data object definitions*,
- $restricted \subseteq A \times D$ is a relation which specifies *restricted data object definitions*,
- $free \subseteq D$ is a relation which specifies *free data object definitions*,
- $condition \in A \mapsto 2^B$ specifies *activity conditions*, where B is a set of partial bindings, i.e., $B = \{f \in D \nrightarrow U | \forall d \in dom(f), f(d) \in dom(d)\}$

such that

- P is acyclic,
- $D = free \cup \{d \in D | \exists a \in A : (a,d) \in mandatory \cup restricted\}$, and
- $free \cap \{d \in D | \exists a \in A : (a,d) \in mandatory \cup restricted\} = \emptyset$.

It is easy to relate Definition 4.1 to the meta model shown in Fig. 1. Set A in Definition 4.1 corresponds to the class *activity definition* in Fig. 1. Set D corresponds to the class *data object definition*. Function *dom* can be considered to be an attribute of the class *data object definition*. Relation P corresponds to the association denoting the precedence relation. Note that we require P to be acyclic, i.e., there are no loops. [3] Functions *mandatory* and *restricted* correspond to the two associations connecting activities and data object definitions. Set *free* corresponds to the association connecting complex case definitions and data object definitions. Note that we do not consider nested case definitions. Therefore, it suffices to consider only one case definition and a set is enough to model free data objects. Free data objects can neither be mandatory nor restricted. Note that a data object definition can be both mandatory and restricted at the same time.

Function *condition* can be seen as an attribute of class *activity definition* in Fig. 1. Each activity definition has a condition which is defined as a set of bindings. A binding is a set of values for specific data objects. An activity can only be executed if the actual values of data objects match at least one of its bindings. If not, the activity is bypassed. Functions *dom* and *condition* provide a very simplistic type system and constraint language. These can be upgraded to more advanced languages. The choice that activities are bypassed if the activity condition evaluates to false is merely chosen for reasons of simplicity. Every activity acts as an AND-join/AND-split [31]. Therefore, sequential and parallel routing are possible by setting the activity conditions to true. Alternative routing, normally specified through XOR-splits and XOR-joins, can be obtained by adding activity conditions such that each activity in one branch either evaluates to true or to false. This style of process modeling corresponds to the routing semantics of InConcert [47]. It is important to note that *activities for which the condition evaluates to false* (*i.e.*, *there is no binding matching the current values*) *are skipped and not blocked*. It is possible to use a less simplistic routing language.

Definition 4.1 is illustrated by the sample case definition shown in Fig. 2. This case definition is formalized as $C1 = (A, P, D, dom, mandatory, restricted, free, condition)$, such that $A = \{A1, A2, A3\}$, $P = \{(A1, A2), (A2, A3)\}$, $D = \{D0, D1, \ldots, D4\}$, and

- $mandatory = \{(A1, D1), (A2, D1), (A3, D1), (A2, D2)\}$,
- $restricted = \{(A3, D3)\}$,
- $free = \{D0, D4\}$.

[3] We do not allow loops. As a result we have a partial order of activities. This is not a fundamental restriction. It is possible to have block structured loops like in MQSeries workflow [32]. However, it is not easy to extend this to the pattern "arbitrary cycles" described in [8]. However, for structured loops the extension is straightforward. In fact, the case handling system FLOW*er* supports this.

Fig. 2 does not specify *dom* and *condition*. Let us assume that $dom(D1) = \{true, false\}$, $dom(D2) = \{red, green, yellow\}$, $dom(D3) = \{1,2,3,4,5,6,7,8,9,10\}$, and $dom(D4) = String$. i.e., $D1$ is a boolean, $D2$ is a color, $D3$ is a number, and $D4$ is some free text. $condition(A1) = \{\}$, which indicates that there is only one possible binding for activity $A1$ and this binding is the empty binding. The empty binding is the function with an empty domain. Therefore, there are no requirements with respect to the values of data objects. This makes sense since $A1$ is the first activity to be executed. $condition(A2) = \{\{(D1, true)\}\}$, which indicates that $A2$ can only be executed if the value of $D1$ is set to *true*. $condition(A3) = \{\{(D2, red)\}, \{(D2, green)\}\}$, which indicates that $A3$ can only be executed if the value of $D2$ is set to *red* or *green*. Suppose that in activity $A1$ data object $D1$ is set to *false* and $D2$ is set to *red*. As a result activity $A2$ is bypassed because $condition(A2)$ does not contain a binding where $D1$ is set to *false*. After skipping $A2$, activity $A3$ becomes enabled. $A3$ is not skipped because there is a binding where $D2$ is set to *red* ($\{(D2, red)\}$). An alternative condition for $A3$ is $condition(A3) = \{\{(D1, true), (D2, red)\}, \{\{(D1, false)\}, (D2, green)\}\}$. This indicates that $A3$ can only be executed if $D1$ is true and $D2$ is red, or $D1$ is false and $D2$ is green. Otherwise $A3$ is bypassed. Note that these examples have only been given to show how conditions can be specified in terms of bindings.

4.2. Dynamics

As a basis for the specification of the dynamic behavior of case handling systems, the behavior of activities has to be defined properly. In this paper, state-transition diagrams are used for this purpose. In a given organization, each case definition is assigned to a particular type of business event, which triggers the instantiation of a case according to the case definition. For example, receiving a message informing an insurance company on a claim is a typical business event. There might be case definitions for which many business events are triggering.

When a case is instantiated, its activities are created. On its creation, an activity is in the initial state. If and when it becomes available for execution, it enters the ready state. When it is selected by the user it starts running. It can either be completed or it can be interrupted. In the latter case, the data entered during the interrupted activity is saved. The activity can be started again, and the data is still available at that time. If all data objects of a given activity are entered, for instance during previous activities, it performs the auto-complete state transition to enter the completed state. Activities may be skipped or bypassed. The user may skip an activity if she decides that it is not required. When due to the evaluation of conditions certain branches are not followed, the activities on that particular branch of the case definition are bypassed.

An important aspect of case handling systems is the ability to re-execute previous activities. This feature is represented by specific redo transitions from the passed, skipped, and completed states. Activities which have been redone can be re-executed. The behavior of activities is shown in Fig. 4.

While activities are an important artifact in case handling, the case is mainly controlled on the basis of states of data objects, associated with the particular case. It is important to stress that not only the life-cycle of activities can be described by states and state transitions, but also data objects. To see this, consider the state transitions that data objects may take as shown in Fig. 5. On the creation of a data object, it adopts the undefined state. Data objects can be defined, either by users filling in forms which represent these data, or they can be defined automatically, for example, by running queries against a database and transferring the result values to the data objects.

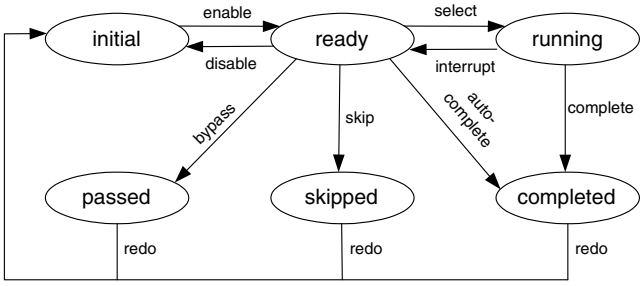

Fig. 4. Dynamic behavior of activities.

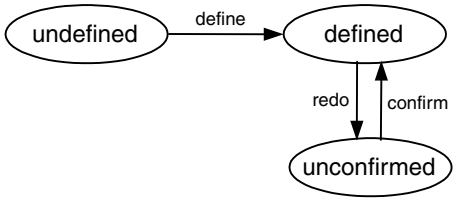

Fig. 5. States of data objects.

Activities for which data objects are mandatory can be redone (cf. the redo role), which results in a state transition of data objects to the unconfirmed state. By confirming the values, data objects re-enter the defined state.

Based on the above considerations, the state space of a case is defined as follows:

Definition 4.2. Let $CD = (A, P, D, dom, mandatory, restricted, free, condition)$ be a case definition. The *case state space* S based on CD is defined as the Cartesian product $S = AS \times DS$ over an *activity state space* AS and a *data state space* DS, such that

- $AS = A \mapsto \{initial, ready, running, completed, passed, skipped\}$, and
- $DS = D \mapsto \{undefined\} \cup (\{defined, unconfirmed\} \times U)$

This definition simply states that the state of a case is characterized by the states of its activities (as characterized by Fig. 4) and the states of data objects (as characterized by Fig. 5). Each data object is either undefined, defined, or—after a redo operation—unconfirmed. In the latter case, a value is stored for the data object.

It is useful to define terms describing the relative order of activities within the context of a given case definition. Given a case definition $CD = (A, P, D, dom, mandatory, restricted, free, condition)$, for each activity $a \in A$

- $preceding(a) = \{a' \in A | (a', a) \in P^+\}$, and
- $subsequent(a) = \{a' \in A | (a, a') \in P^+\}$.

where $P^+ = \cup_{i>0} P^i$ is the non-reflexive transitive closure of P.

Case handling systems make use of case definitions to guide users in handling cases. In order to do that, the system has to make sure that a given activity is flagged ready for execution if and only if the preconditions of that activity are met. To be able to specify if an activity should be executed or bypassed, we use the following auxiliary function. Let $CD = (A, P, D, dom, mandatory, restricted, free, condition)$ be a case definition and $S = AS \times DS$ its state space. Function $\alpha \in DS \mapsto (D \nrightarrow U)$ maps elements of the data state space onto sets of defined data objects and their values, i.e., α filters out data objects which are undefined or unconfirmed. α can be specified as follows: For any $ds \in DS$: $\alpha(ds) := \{(d, v) \in D \times U \mid ds(d) = (defined, v)\}$. Using this function, we can define whether an activity $a \in A$ should be executed considering a data state $ds \in DS$: $C_{pre}(a, ds) := \exists f \in condition(a) : f \subseteq \alpha(ds)$. $C_{pre}(a, ds)$ is called the precondition of activity a in data state ds. Note that $C_{pre} \in (A \times DS) \mapsto \mathbb{B}$. Note that if this condition evaluates to be true, a user with the proper role can select the activity for execution. If the condition evaluates to be false, the activity is bypassed. Again we would like to stress that activities may be bypassed but not blocked like in most other languages.

In addition to a precondition which depends on the data state, there is also a postcondition depending on the data state. $C_{post} \in (A \times DS) \mapsto \mathbb{B}$ is an auxiliary function for specifying postconditions. For each $a \in A$ and $ds \in DS$, $C_{post}(a, ds) := \{d \in D \mid (a, d) \in mandatory\} \subseteq dom(\alpha(ds))$ is the postcondition of activity a in data state ds.

Functions C_{pre} and C_{post} only focus on the data state $ds \in DS$. Clearly, the data state is not sufficient to determine the dynamics, also the activity state $as \in AS$, the causal relations specified by P, and the state-transition diagrams shown in Figs. 4 and 5 matter. To specify the semantics of case handling we augment the state transitions shown in Fig. 4 with rules specified using an event condition action (ECA) style of formalization [45]. Each state transition shown in Fig. 4 is described by a rule of the following form: **ON** *event*, **IF** *condition*, **THEN** *action*. The event describes the trigger to evaluate the rule and typically corresponds to a user action. If there is no external event needed to trigger the rule (i.e., a system trigger), this part of the rule is omitted. The condition is a boolean expression in terms of the state of the case, i.e., the activity state ($as \in AS$) and the data state ($ds \in DS$). The action is a state transition in the state-transition diagram. Using such ECA-rules, the semantics are defined as follows.

Definition 4.3. Let $CD = (A, P, D, dom, mandatory, restricted, free, condition)$ be a case definition, $a \in A$ an activity, $as \in AS$ the activity state, and $ds \in DS$ the data state. The state transitions shown in Fig. 4 are defined by the following ECA-rules.

- **IF** $\forall a' \in preceding(a) : as(a') \in \{\text{passed}, \text{skipped}, \text{completed}\}$
  THEN $enable(a, as, ds)$
- **IF** $\exists a' \in preceding(a) : as(a') \notin \{\text{passed}, \text{skipped}, \text{completed}\}$
  THEN $disable(a, as, ds)$
- **ON** user trigger (an actor with the proper execute role selects the activity)
  IF $C_{pre}(a, ds)$
  THEN $select(a, as, ds)$
- **ON** user trigger (activity is interrupted by the actor working on the activity)
  IF true
  THEN $interrupt(a, as, ds)$

- **ON** user trigger (activity is completed by the actor working on the activity)
  IF $C_{post}(a, ds)$
  THEN $complete(a, as, ds)$
- **IF** $C_{pre}(a, ds) \wedge C_{post}(a, ds)$
  THEN $auto\_complete(a, as, ds)$
- **ON** user trigger (activity is skipped by an actor with the proper skip role)
  IF $C_{pre}(a, ds)$
  THEN $skip(a, as, ds)$
- **IF** $\neg C_{pre}(a, ds)$
  THEN $bypass(a, as, ds)$
- **ON** user trigger (activity is redone by an actor with the proper redo role)
  IF $\forall a' \in subsequent(a) : as(a') \in \{\text{initial}, \text{ready}\}$
  THEN $redo(a, as, ds)$

The ECA rules should be interpreted in the context of the state-transition diagram shown in Fig. 4. A rule can only be applied if the corresponding activity is in the proper state, e.g., action $bypass(a, as, ds)$ corresponds to a state transition of state ready to state passed and, therefore, can only be executed if activity a is in state ready. Most of the rules are fairly straightforward. The only rule which deserves some explanation is the last one, $redo(a, as, ds)$. To redo an activity all *subsequent* activities should either be in state initial or ready or also rolled back. Therefore, one should first roll back activities whose subsequent activities are ready or initial and then recursively roll back the other activities. Note that it is possible that a direct predecessor of an activity that is in state ready can be rolled back. If this is the case, action $disable(a, as, ds)$ automatically puts the predecessor in state initial.

Definition 4.3 only relates to the state-transition diagram shown in Fig. 4. In the next definition we give similar rules for the state-transition diagram shown in Fig. 5.

Definition 4.4. Let $CD = (A, P, D, dom, mandatory, restricted, free, condition)$ be a case definition, $d \in D$ a data object, $as \in AS$ the activity state, and $ds \in DS$ the data state. The state transitions shown in Fig. 5 are defined by the following ECA-rules.

- **ON** user trigger (an actor enters the value of a data object in a form)
  IF $(\exists a \in A : (a, d) \in restricted) \Rightarrow (\exists a \in A : (a, d) \in restricted \wedge as(a) = running)$
  THEN $define(d, as, ds)$
- **ON** system trigger (if an activity is redone all data elements associated to the activity are triggered)
  IF true
  THEN $redo(d, as, ds)$
- **ON** user trigger (the value of a data object is confirmed by an actor having access to some form)
  IF $(\exists a \in A : (a, d) \in restricted) \Rightarrow (\exists a \in A : (a, d) \in restricted \wedge as(a) = running)$
  THEN $confirm(d, as, ds)$

It is interesting to note that the state-transitions in Fig. 5 are relatively independent of the states of activities. This is the essence of case handling, the data objects are leading and data values may

be entered at various places. Only restricted data objects are closely bound to activities. This is reflected in the conditions given in Definition 4.4.

4.3. Other aspects

The formalization given in terms of the state-transition diagrams and the ECA rules only partially incorporates aspects such as forms and roles. Therefore, we discuss the relationships between these aspects and Definitions 4.1–4.3, and 4.4.

Form definitions are linked to activity definitions and complex case definitions. Typically, if $(a,d) \in mandatory$, then data object d also appears in the form linked to activity a. Note that a form linked to an activity may contain entries for data objects that are not mandatory. These additional entries may be used to enter data which is needed in subsequent activities or to view and modify data produced in preceding activities. The additional entries increase flexibility by decoupling data objects and activities. There may even be forms which are not linked to any activity. Forms do not determine whether a data object is mandatory, restricted, or free. This is a matter between activities and data objects. Given the limited impact of forms on the dynamics of case handling, we abstracted from this aspect.

Roles are linked to activities. We distinguish at least the following three role types: exec, skip and redo. These roles are mentioned in the event part of the ECA rules given in Definition 4.3 and 4.4. For example, it is only possible to skip an activity if the event that leads to action $skip(a,as,ds)$ is generated by an actor that has the skip role.

An issue that was not addressed is the separation between work distribution and authorization. In traditional workflow management systems work distribution and authorization coincide. For case handling we propose the query mechanism mentioned before. Users can simply state an ad hoc query or use a predefined query. The query "Select all cases for which there is an activity in state ready which has an execute role R" can be used to emulate the traditional in-tray. The query mechanism is used to give an actor a *handle* to a case and not to a specific activity. Once an actor has a handle to a case, she can select activities that are in state ready. Note that authorization is governed by the exec, skip and redo roles. Work distribution is governed by the query mechanism.

5. FLOW*er*

In this section we introduce a concrete case handling product: FLOW*er*. FLOW*er* [5,12,13] is Pallas Athena's case handling product. FLOW*er* is consistent with the case handling meta model (cf. Section 3) and the formal framework (cf. Section 4). However, FLOW*er* offers much more features than discussed in the previous sections. For example, Section 4 assumes a rather basic control flow model where eventually all activities are either bypassed, skipped, or completed. In this basic model it is not possible to select one alternative branch, have multiple instances, deferred choice, etc. [8]. As a result, Section 4 presents only a simplification of the actual functionality of FLOW*er*. Note that the goal of this paper is to show the essence of case handling and not a concrete product. Nevertheless, we think it is interesting a see a concrete application of FLOW*er* to illustrate the case handling paradigm.

FLOW*er* consists of a number of components: FLOW*er* Studio, FLOW*er* Case Guide, FLOW*er* Configuration Management (CFM), FLOW*er* Integration Facility, and FLOW*er* Management Information and Case History Logging. In this paper, we limit ourselves to FLOW*er* Studio and FLOW*er* Case Guide. FLOW*er* Studio is the graphical design environment. It is used during build-time to define case definitions, consisting of activities, precedences, data objects, roles, and forms. FLOW*er* Case Guide is the client application which is used to handle individual cases.

Now we consider a fictive insurance company's process for handling claims for motor car damage. Fig. 6 shows a top-level view of the workflow process *MotorClaim* in FLOW*er* Studio. The right-hand side of Fig. 6 shows a graphical representation of the process. The left-hand side shows a list of data object definitions. The left-hand side of the window can also be used to list all form definitions, mappings (to connect to external information sources) and complex case definitions (subprocesses). As Fig. 6 shows the case handling process starts with the creation of a case (activity *Case_Creation*), followed by the activity *Claim_Start*. Activity *Claim_Start* is linked to a form which enables the user to enter the claim data and the scanned hand-written form supplied by the claimant. Both data objects are restricted, i.e., they can only be entered in this step in the process. After completing the form associated to activity *Claim_Start* the subprocess *Register_Claim* is started. Note that this corresponds to a complex case definition in terms of our meta model (cf. Fig. 1). Complex case definitions are named plans in FLOW*er*. *Register_Claim* is a so-called static plan which means that it does not involve any choices and is instantiated only once. The top-level view of *Register_Claim* is shown in Fig. 7. *Register_Claim* consists of a number of activities which all need to be executed and each of these activities corresponds to obtaining certain data objects. After completing *Register_Claim*, four complex case definitions are handled in parallel: *Get_Medical_Report*, *Get_Police_Record*, *Assign_Loss_Adjuster*, and *Witness_Statements*. *Get_Medical_Report*, *Get_Police_Record*, and *Assign_Loss_Adjuster* correspond to subprocesses which start with a system choice and are named system decision plans. Each of these subprocesses contains several activities. A detailed description of these subprocesses is beyond the scope of the

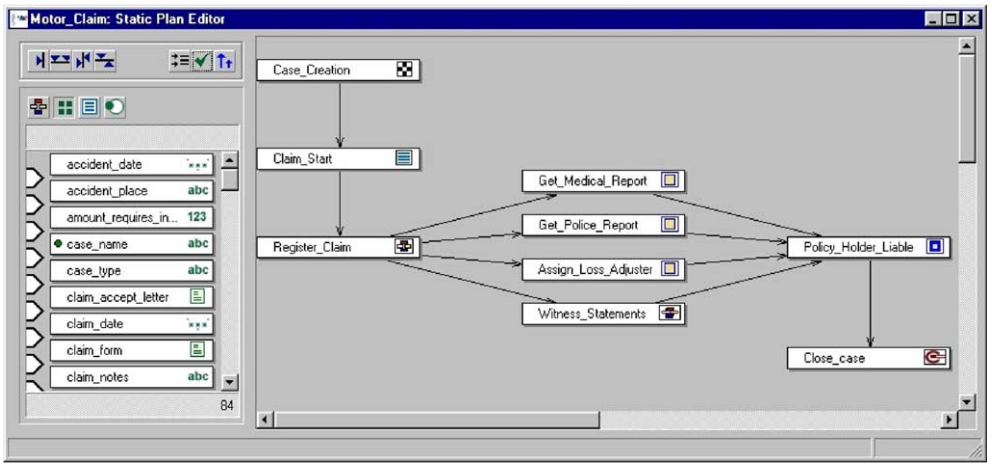

Fig. 6. Complex case definition *MotorClaim*.

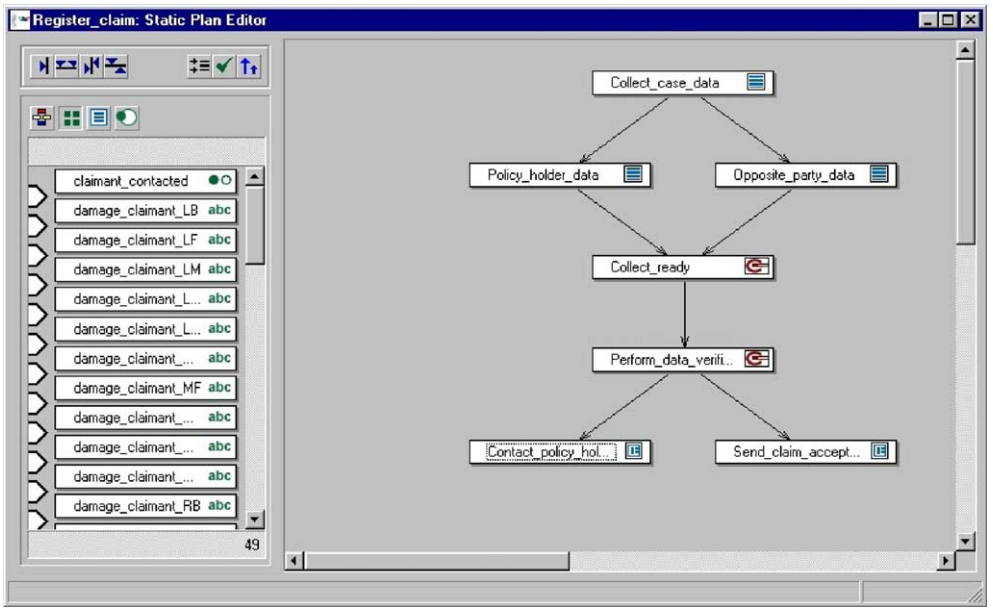

Fig. 7. Complex case definition *Register_Claim*.

paper. The same holds for the processing of witness statements. However, the complex case definition *Witness_Statements* is a so-called dynamic subplan. This means that it can be instantiated multiple times and each of these instances is handled in parallel. A dynamic subplan can have the following attributes: *Expansion name*, *Minimum instances*, and *Max expansions*. The attribute *Expansion name* is used to identify each instance. For the subplan *Witness_Statements* the name of the witness is used. The attribute *Minimum instances* is used to specify how many instances should be created (in this case the number of eye witnesses specified by the data object *nr_witnesses* entered in *Register_Claim*). The attribute *Max expansions* is used to set an upper limit for the number of instances (in this case 5; note that new instances can be created on-the-fly).

After completing *Get_Medical_Report*, *Get_Police_Record*, *Assign_Loss_Adjuster*, and *Witness_Statements*, complex case definition *Policy_Holder_Liable* is executed. This subprocess starts with a user decision and is therefore named a user decision plan. *Policy_Holder_Liable* contains seven activities. Again details are omitted.

The case definition of MotorClaim comprises 173 data object definitions. This number shows the relevance of data. Each data object has a name and a type and is linked to a plan (i.e., a complex case definition). The left-hand side of Fig. 8 shows these attributes for the data object definition *claimant_contacted*. This is a boolean data object indicating whether the policy holder has been contacted. Initially this data object is set to false. As the right-hand side of Fig. 8 shows, *claimant_contacted* is restricted to activity *Contact_policy_holder*. This activity is part of the complex case definition *Register_Claim* shown in Fig. 7. Note that one data object definition can be restricted to multiple activity definitions and that one activity definition can have multiple restricted data object definitions. This is consistent with the cardinalities of the association *restricted* shown in Fig. 1. Mandatory data objects are specified when defining an activity. Fig. 9

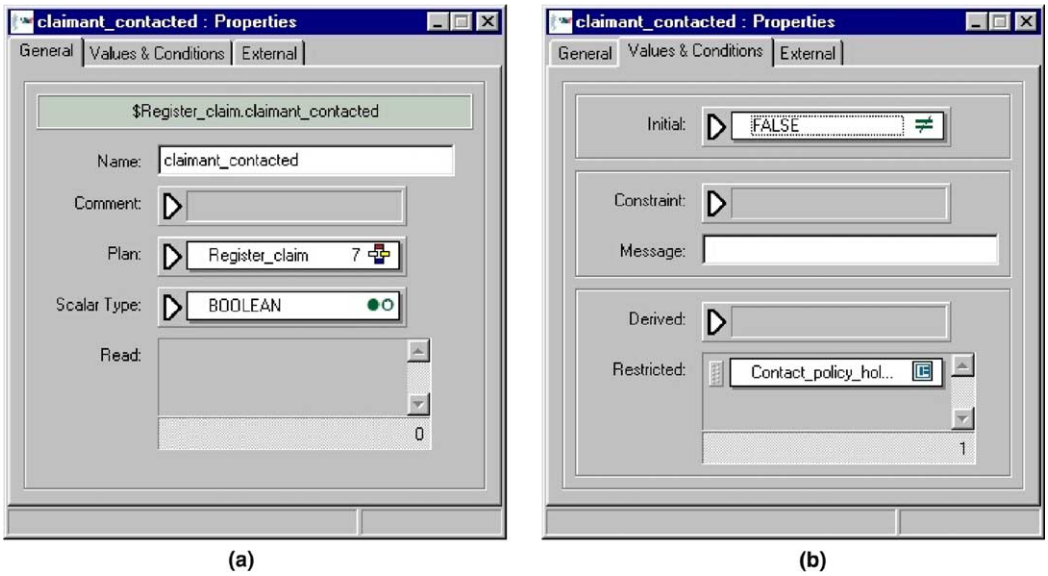

Fig. 8. Attributes of the data object definition *claimant_contacted*.

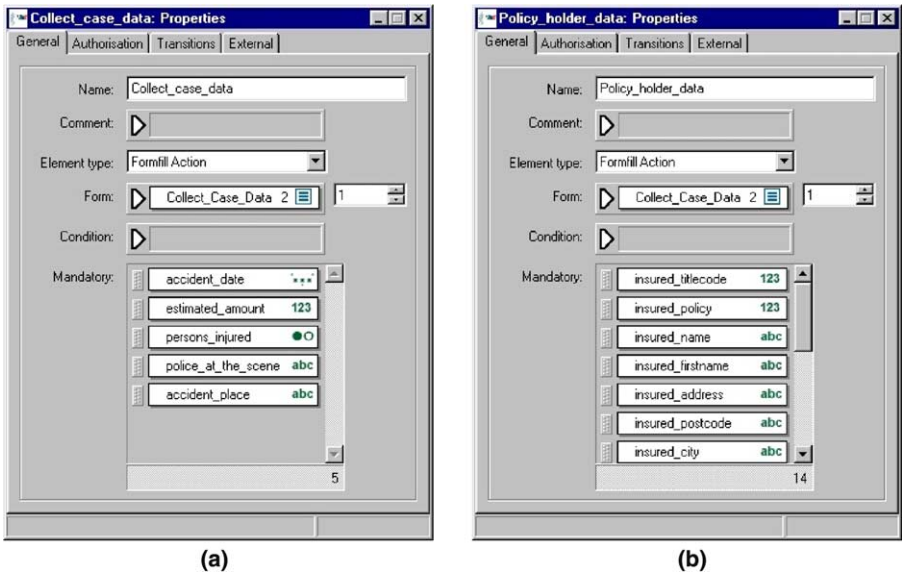

Fig. 9. Properties of activities, including specification of mandatory data objects.

shows two activities and the corresponding lists of mandatory data objects. For example, data object definition *accident_date* is mandatory for activity definition *Collect_case_data*. All data object definitions are linked to a specific complex case definition (i.e., including restricted and mandatory data elements). For example, the left-hand side of Fig. 8 shows that *claimant_contacted* is linked to plan *Register_Claim*. This is consistent with the meta model which identifies the association *free* (cf. Fig. 1) which links complex case definitions and data object definitions. However,

the realization in FLOW*er* implies that all mandatory and restricted data objects are also linked to a complex case definition (i.e., plan).

The case definition of MotorClaim comprises 21 form definitions. One form definition can be linked to many activity definitions. For example, form definition *Collect_Case_Data* is linked to the first four activities of *Register_Claim*. Fig. 9 shows two activity definitions sharing this form. Let us focus on the first three steps of *Register_Claim* (Fig. 7). Activity definition *Collect_case_data* has 5 mandatory data object definitions (accident date, persons injured, etc.). Activity definition *Policy_holder_data* has 14 mandatory data object definitions (name of policy holder, policy number, etc.). Activity definition *Opposite_party_data* has 10 mandatory data object definitions (name of opposite party, address, etc.). There is no overlap between these mandatory data objects. However, form definition *Collect_Case_Data* includes all these data objects since the form is shared among these activities. This means that when a worker is executing the first step in the process (i.e., activity *Collect_case_data*), she will see information relevant for subsequent steps in the process. Moreover, the worker can already enter data and this way implicitly execute subsequent steps. By entering the 5 + 14 + 10 = 29 mandatory data objects mentioned before, the first three steps are executed through filling out a single form. This example demonstrates the essence of case handling: The focus is on the whole case rather a single work-item and data objects rather than control-flow constructs are driving the workflow.

As Fig. 10 shows, six roles are relevant for the *MotorClaim* case definition: *nobody*, *Manager*, *Supervisor*, *Claim_adjuster*, *Doctor*, and *Data_collector*. The arcs in the role graph correspond to the *is_a* association shown in Fig. 1. Note that *nobody* is the most powerful role. If no actors are assigned this role, it can be used to disable undesirable skip or redo actions as was explained

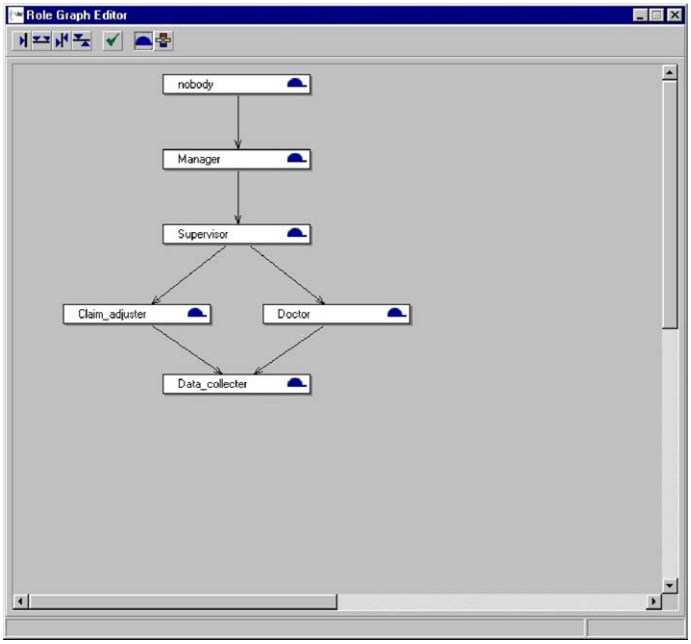

Fig. 10. Role graph editor of Studio showing the six roles involved.

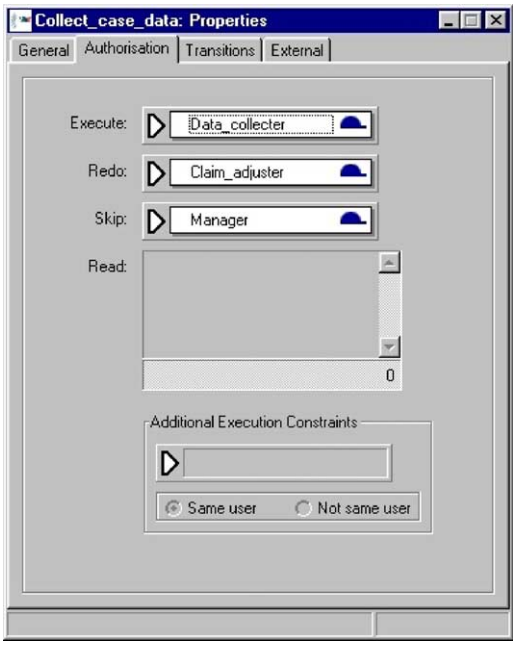

Fig. 11. The execute, redo, and skip roles of *Collect_case_data*.

in Section 2. The role *Data_collector* is the weakest role and this role can be fulfilled by anybody having any of the six roles shown in Fig. 10. Each activity definition has three types of roles assigned to it. Fig. 11 shows the execute, redo, and skip roles of *Collect_case_data*. *Collect_case_data* can be executed by workers with at least the role *Data_collector* (i.e., all actors having any of the six roles), it can be redone by workers with at least the role *Claim_adjuster* (i.e., all actors except the ones just having the *Doctor* or *Data_collector*) role, and it can be skipped by workers with at least the role *Manager* (i.e., all actors with either the role *Manager* or *nobody*).

Figs. 6–11 show windows of the design tool Studio. Actors (i.e., workers) access cases through the so-called FLOW*er* Case Guide. Access to cases is limited by the associated roles. Note that FLOW*er* supports the separation of authorization and work distribution. The role mechanism is used for authorization. Work distribution is supported through a query mechanism as explained in Section 2.

Fig. 12 shows the Case Guide showing the state of a case of type *MotorClaim*. The case guide shows the whole case. The left-hand-side shows the hierarchy of the case definition. The right-hand side of the Case Guide shown in Fig. 12 is divided into three parts. The top part is used for navigation. The bottom part is used to access forms which are independent of activities, e.g., form *Case Overview* can be opened at any time and shows information about letters sent, letters received, the accident form, etc. In the middle part of the right-hand side for the window, the so-called *wavefront* is shown. The wavefront is the most essential piece of information provided by the Case Guide since it shows the state of the case in terms of activities that have been executed or skipped, activities that are enabled, and activities that are not (yet) enabled. The wavefront provides a time line. Activity *Claim_start* is on the right of this time line indicating that it has been executed. Static plan

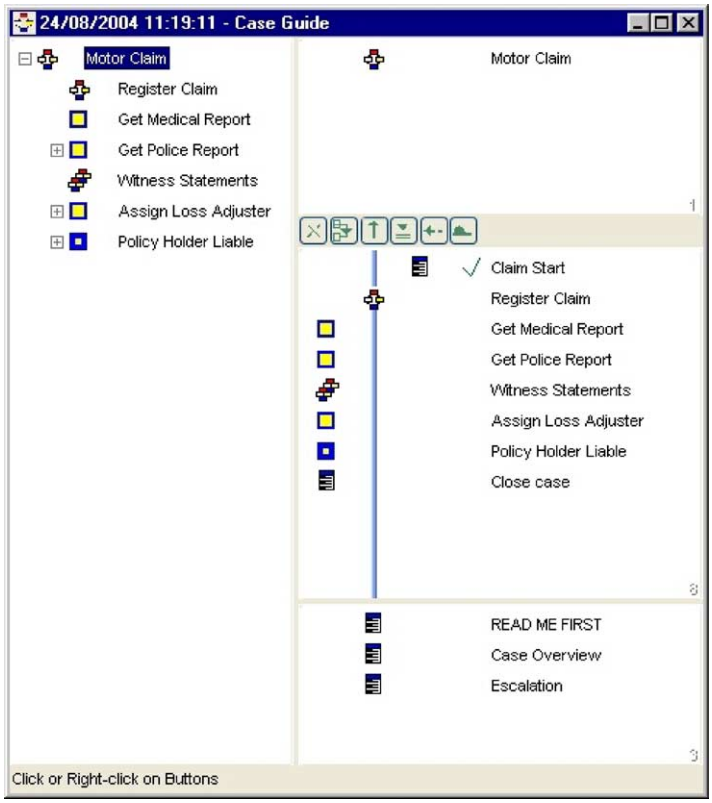

Fig. 12. The FLOW*er* Case Guide.

(i.e., subprocess/complex case definition) *Register_Claim* is on the time line indicating that it is ready to be executed. *Get_Medical_Report* and the other plans/activities at the top level are on the left of the time line indicating that they are not (yet) enabled. By double clicking the icon of *Register_Claim*, the wavefront for the activities/plans inside *Register_Claim* is shown. By double-clicking an activity, the execution of the corresponding activity starts. If the first activity of *Register_Claim* (i.e., *Collect_case_data*) is started, the form shown in Fig. 13 is opened. This form, also named *Collect_Case_Data*, consists of two pages. Fig. 13 only shows the first page. The first six data objects shown in the form correspond to the activity *Collect_case_data*. The data objects under "INSURO client" correspond to activity *Policy_holder_data* and the data objects under "Opposite party" correspond to activity *Opposite_party_data*. The form *Collect_Case_Data* is linked to these three activities, i.e., a single form is shared among multiple activities. However, whether data objects are mandatory or restricted depends on the current activity. Note that as indicated before all three activities can be performed through a single form, i.e., there is no need to open and close forms in-between activities. However, a worker can fill out only the top part of the form *Collect_Case_Data* and thus only executed the first step in *Register_Claim*.

In this section, we have shown an application of case handling using FLOW*er*. The application is fairly straightforward. However, even rather straightforward workflow processes may involve many data objects and activities. The MotorClaim applications consists of 8 complex case

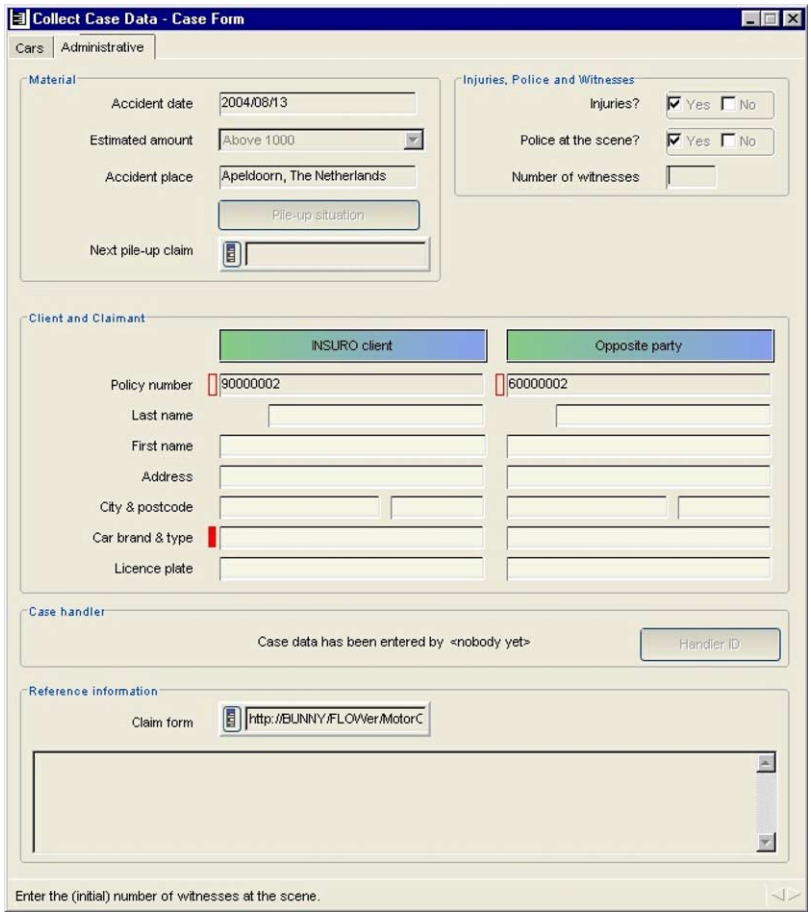

Fig. 13. The form associated with some of the activities in *Register_Claim*.

definitions, 30 activity definitions, 173 data object definitions, 21 forms, and 6 roles. Unlike typical workflow solutions, data is not hidden inside applications. The fact that a considerable number of data objects are identified and used for guiding workers through case executions shows that the process is primarily data-driven and thus a nice illustration of the case handling paradigm.

6. Applications of case handling

In this section we briefly discuss practical implementations of case handling systems based on FLOW*er*. At this point in time several Dutch organizations are switching from a traditional workflow management system to FLOW*er*. In many cases the switch is triggered by the problems addressed in the introduction. An example of such an organization is the UWV. The Employee Insurance Implementing Body (Uitvoering Werknemersverzekeringen, or UWV) is responsible for the implementation of employee insurance schemes, such as the sickness insurance scheme (ZW), the national health insurance scheme (ZFW), the unemployment insurance scheme

(WW) and the occupational disability insurance scheme (WAO). The UWV levies the contributions under these schemes, assesses benefit applications and sees to the payment of benefits. UWV is a new organization which joins former organizations (uitvoeringsinstellingen, or uvi's) such as Guo, Gak, Cadans, Uszo, Bouwnijverheid, and Lisv. Formally the organization was created on 1-1-2002. The goal of joining these organizations into the UWV is twofold: improve quality and reduce costs. To achieve this goal, the merger triggered the development of a common ICT strategy and standards. For workflow management and case handling within the UWV, FLOW*er* was selected as the standard product. One of the first applications of FLOW*er* was in the Complaints and Appeals Department of UWV/Gak. A system based on FLOW*er* runs within all 25 branch offices (1000 users) of the UWV/Gak and handles about 110.000 complaints and 15.000 appeals per year. The processes supported by this system are complex, need to deal with many exceptions, and involve many documents. Since FLOW*er* is a rather new tool and the organizations applying FLOW*er* are not eager to provide detailed information about their processes (e.g., for reasons of confidentiality), we cannot provide any details of current implementations. Instead we conclude with a list of typical application domains of a tool like FLOW*er*:

- Payment institutions (issuing payments, handling complaints and appeals),
- Banks and insurance companies (credit facilities, claims processing),
- Government bodies (processing vertical products),
- Telecommunications (client and contract administration),
- Housing corporations (real estate administration),
- Educational institutions (student and course administration),
- Health care (patient registration and administrative processing),
- Police (supporting police field work),
- Courts of law (writs, summonses), and
- IT companies and departments (incidents, requests for changes).

It is important to note that FLOW*er* has been applied in *each* of these application domains, i.e., it is not only a list of possible applications: It lists areas where case handling has been applied using FLOW*er*.

7. Related work

As was mentioned in the introduction, many researchers have recently addressed the issue of workflow flexibility; a number of workshop reports, edited books, and special issues of journals were devoted to this topic, e.g., [6,9,16,23,27,29,30]. Agostini and De Michelis [11] 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., [15], give concrete adaptation rules. Some authors even state that "workflow change is a workflow" [19]. Several authors propose a more declarative style of specifying workflows, for example the Vortex paradigm [25]. Approaches like [16,23,27] use the metaphor of an active document. 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 [24,41]. Herrmann [24] seeks a solution by using semi-structured workflow models. Reijers

et al. [41] propose a product-driven approach to emphasize the role of data objects in the design of workflows. The latter approach can easily be combined with the case handling paradigm.

This paper builds on [5] which introduced the basic idea of case handling without providing a meta model, formalization, realistic examples, etc. In [10] we presented the application of case handling in a concrete project. In [40] we put this work in the context of traditional workflow systems.

Schuschel et al. introduce an integrated approach for process planning and coordination, based on planning algorithms developed by the artificial intelligence community. Pre conditions and post conditions are used to derive goals based on a current situation. While this work is on a conceptual level and no implementation of the ideas is given, it introduces planning as a vehicle for flexible process modeling, which is strongly related to case handling [42].

Increasingly, agent technology is used to build workflow management systems [49,36,37,48]. The agent architecture allows for additional flexibility. Despite the many agent-based workflow prototypes, the authors are unaware of any commercial applications of agents in the workflow domain. The approach based on agents is related to the work on proclets [4], where complex workflows are partitioned into interacting simple workflows using the ideas of [50].

Vendors of workflow management systems have also been struggling with flexibility issues. Systems such as InConcert (TIBCO, [47]) 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. We know of only few systems that claim to support case handling. We elaborated on FLOW*er* developed by Pallas Athena [5,13,12]. The functionality of FLOW*er* is more-or-less consistent with the meta model and formalizations given in this paper. Vectus (London-Bridge/Hatton Blue) and the Staffware Case Manager [44] are two other systems also aiming at case handling. Initially the focus of Vectus was on workflow support for legal applications. Since London-Bridge has acquired Hatton Blue, the focus has shifted to Customer Relationship Management (CRM). The Staffware Case Manager (SCM) extends the Staffware workflow management system with case handling functionality. The SCM can be seen as a layer between the workflow management system and the applications. The SCM allows for easy access to case related documents and is able to refine Staffware steps (i.e., workflow activities) into more fine-grained tasks. In essence, the resulting approach is still process-driven and not data-driven. The only way to get to the SCM is through the standard work-queue mechanism of the Staffware workflow management system. Therefore, the meta model and the formal framework for case handling systems do not apply to the SCM. Similar to SCM is the COSA Activity Manager (CAM) [43]. Both SCM and CAM are based on the Activity Manager of BPi [14].

Besides contemplating concrete case handling systems, it is interesting to consider case handling in the context of Computer Supported Cooperative Work and, more specifically, groupware systems [18,28]. An interesting classification of collaborative technologies is given in [17]. There Ellis presents a taxonomy dividing collaborative technologies into four classes of functionality:

- *Keepers* support the access and change to shared artifacts. Typical issues that are of primary concern to keepers are access control, versioning, backup, recovery, and concurrency control. Examples of keepers include the vault in a Product Data Management (PDM) system, a repository with drawings in a CAD/CAM system, and a multi media database system.

- *Coordinators* are concerned with the ordering and synchronization of individual activities that make up the whole process. Typical issues addressed by coordinators are process design, process enactment, enabling of activities, and progress monitoring. The key functionality of a workflow management system is playing the role of coordinator.
- *Communicators* are concerned with explicit communication between participants in collaborative endeavors. Typical examples are electronic mail systems and video conferencing systems, and basic issues that need to be addressed are message passing (broadcast, multicast, etc.), communication protocols, and conversation management.
- *Team-agents* are specialized domain-specific pieces of functionality. A team agent is typically a system acting on behalf of a specific person or group and executing a specific task. Examples include an electronic agenda and a meeting scheduler.

The functionality of workflow management systems is usually limited to the coordinator role. Clearly a case handling system supports the keeper, coordinator, and communicator roles. A case handling system extends the traditional workflow management system with the keeper role (data objects are under the control of the case handling system) and better support for the communicator role (the separation for work distribution and authorization and the ability to attach semi-structured information to cases). Groupware systems (i.e., excluding workflow technology) tend to be weak on the coordination dimension, and stronger on the keeper, communicator, and team-agent functions. Many groupware systems provide various kinds of support for group decision-making, but they do not have any explicit notion of workflow processes. Lotus Notes combined with Domino Workflow [38] forms an exception, which does provide all four functions. Therefore, it is interesting to compare Lotus Notes/Domino Workflow with the case handling paradigm presented in this paper. Domino Workflow supports the so-called binder concept which is a logical structure grouping documents related to one case. Moreover, the keeper, coordinator and communicator roles are truly integrated. However, the routing between activities is comparable to a traditional workflow management system, i.e., the approach is process-driven rather than data-driven. Therefore, Lotus Notes/Domino Workflow should not be considered a case handling system. Nevertheless, Lotus Notes/Domino Workflow is targeting similar problems.

8. Conclusions

This paper introduced case handling as a new paradigm for supporting flexible business processes. Using the ''Blind Surgeon Metaphor'' we introduced case handling and highlighted the problems with respect to contemporary workflow technology. In Section 2 we compared case handling with traditional workflow management. Striking differences are the difference in focus (on the whole case rather than an individual work-item) and the difference in driving force (also data-driven, not only process-driven). To precisely define the case handling metaphor we provided a meta model at both the schema level and the instance level, and a formal mathematical framework specifying the dynamics of case handling in terms of state-transition diagrams and ECA rules. Finally, we presented an example and discussed related work and products.

In the introduction of this paper, we identified four problems. These problems are addressed by case handling in the following way:

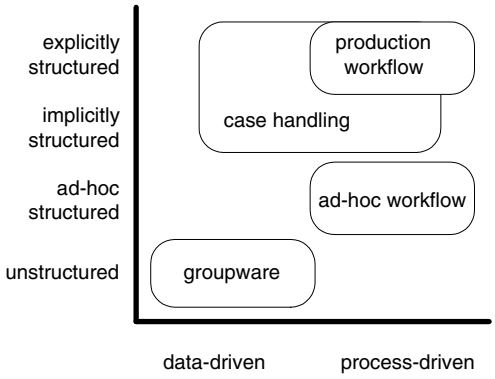

Fig. 14. Positioning case handling.

- context tunneling is avoided by providing all information available (i.e., present the case as a whole rather than showing just bits and pieces),
- activities are enabled on the basis of the information available rather than the activities already executed,
- work distribution is separated from authorization and additional types of roles (i.e., not just the execute role) are added,
- workers are allowed to view and add/modify data before or after the corresponding activities have been executed (e.g., information can be entered the moment it becomes available).

To conclude this paper, we position case handling in the context of groupware, production workflow, and ad hoc workflow using Fig. 14. Traditional groupware products like Lotus Notes and MS Exchange and production workflow systems like Staffware and MQSeries Workflow form two ends of a spectrum. As Fig. 14 shows, traditional groupware products are data-driven (focus on the sharing of information rather than the process) and support only unstructured processes. Note that Lotus Notes and Exchange are not "process-aware" (unless components like Domino Workflow are added). Production workflow are process-aware and aim at structured processes. In order to enact a workflow using a production workflow system one needs to explicitly specify all possible routes. If something is not explicitly specified at design time, it is not possible. Ad hoc workflow management systems like InConcert (TIBCO), Ensemble (Filenet), and TeamWARE Flow (TeamWARE Group) allow for the creation and modification of workflow processes during the execution of the processes. Each case has a private process model and therefore the traditional problems encountered when changing a workflow specification can be avoided. Ad hoc workflow management systems allow for a lot of flexibility. The workflow management system InConcert even allows the user to initiate a case having an empty process model. When the case is handled, the workflow model is extended to reflect the work conducted. Another possibility is to start using a template. The moment a case is initiated, the corresponding process model is instantiated using a template. After instantiation, the case has a private copy of the template, which can be modified while the process is running. InConcert also supports "workflow design by discovery": The routing of any completed workflow instance can be used to create a new template. This way actual workflow executions can be used to create workflow process definitions. Fig. 14 shows that

ad hoc workflow management systems like InConcert are process-driven and ad hoc structured. Note that, per case, there has to be an explicit process model. Although interesting, the practical relevance of ad hoc workflow is limited since it assumes that workers can modify models at runtime. Although flexible, this poses many problems ranging from unauthorized actions to incomplete cases.

As Fig. 14 shows, case handling should be positioned in-between groupware, production workflow, and ad hoc workflow. Note that case handling is both process driven and data driven. On the one hand, it is possible to create data-driven workflows by using mandatory data objects. On the other hand, it is possible to define causal dependencies like in a traditional workflow system. Using a data-driven approach it is possible to allow for many routes without explicitly specifying them. Moreover, additional roles like skip and redo also enable implicit routes through the workflow. If each task has "everyone" as its execute, skip, and redo role, there are hardly any constraints. If all data elements are mandatory and restricted and each task has "nobody" as its skip and redo role, one obtains the functionality comparable to a traditional workflow system. Hence case handling encompasses the traditional workflow paradigm.

Clearly there is a trade-off between support and flexibility. Systems that offer a lot of support tend to be inflexible (e.g., production workflow). Systems that offer a lot of flexibility tend to offer less support to the process (e.g., groupware products). At a first glance, support and flexibility may even seem to be contrasting. As demonstrated in this paper, the case handling paradigm offers an interesting balance between support and flexibility. Nevertheless, future research will be aiming at further developing the case handling concepts and empirical proof of the statements made in this paper. Moreover, the case handling paradigm does not solve all problems related to change. Note that structural changes will still lead to problems such as the dynamic change bug [3,20,39,51].

Acknowledgement

We thank all the people from the FLOW*er* development team of Pallas Athena for their insights in case handling and their support with respect to the use of FLOW*er*. Specifically we would like to thank Paul Berens for sharing his knowledge of case handling with us. We also thank Eric Verbeek for his technical support in using FLOW*er* and Moniek Stoffele for applying the case handling concept to the building processes of Heijmans [10]. Finally, we thank the three reviewers for their useful comments.

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) (1998) 21–66.
[2] W.M.P. van der Aalst, On the automatic generation of workflow processes based on product structures, Computers in Industry 39 (1999) 97–111.
[3] W.M.P. van der Aalst, Exterminating the dynamic change bug: A concrete approach to support workflow change, Information Systems Frontiers 3 (3) (2001) 297–317.
[4] W.M.P. van der Aalst, P. Barthelmess, C.A. Ellis, J. Wainer, Proclets: A framework for lightweight interacting workflow processes, International Journal of Cooperative Information Systems 10 (4) (2001) 443–482.

[5] W.M.P. van der Aalst, P.J.S. Berens, Beyond workflow management: Product-driven case handling, in: S. Ellis, T. Rodden, I. Zigurs (Eds.), International ACM SIGGROUP Conference on Supporting Group Work (GROUP 2001), ACM Press, New York, 2001, pp. 42–51.
[6] W.M.P. van der Aalst, J. Desel, A. Oberweis (Eds.), Business Process Management: Models, Techniques, and Empirical Studies, Volume 1806 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2000.
[7] W.M.P. van der Aalst, K.M. van Hee, Workflow Management: Models, Methods, and Systems, MIT press, Cambridge, MA, 2002.
[8] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, A.P. Barros, Workflow patterns, Distributed and Parallel Databases 14 (1) (2003) 5–51.
[9] W.M.P. van der Aalst, S. Jablonski, Dealing with workflow change: Identification of issues and solutions, International Journal of Computer Systems, Science, and Engineering 15 (5) (2000) 267–276.
[10] W.M.P. van der Aalst, M. Stoffele, J.W.F. Wamelink, Case handling in construction, Automation in Construction 12 (3) (2003) 303–320.
[11] A. Agostini, G. De Michelis, Improving flexibility of workflow management systems, in: W.M.P. van der Aalst, J. Desel, A. Oberweis (Eds.), Business Process Management: Models, Techniques, and Empirical Studies, Volume 1806 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2000, pp. 218–234.
[12] Pallas Athena, Case Handling with FLOWer: Beyond Workflow, Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
[13] Pallas Athena, Flower User Manual, Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
[14] BPi. Activity Manager: Standard Program–Standard Forms (Version 1.2), Workflow Management Solutions, Oosterbeek, The Netherlands, 2002.
[15] F. Casati, S. Ceri, B. Pernici, G. Pozzi, Workflow evolution, in: Proceedings of ER'96, Cottubus, Germany, October 1996, pp. 438–455.
[16] P. Dourish, W.K. Edwards, J. Howell, A. LaMarca, J. Lamping, K. Petersen, M. Salisbury, D. Terry, J. Thornton, A programming model for active documents, in: Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology, ACM Press, 2000, pp. 41–50.
[17] C.A. Ellis. An Evaluation Framework for Collaborative Systems. Technical Report, CU-CS-901-00, Department of Computer Science, University of Colorado, Boulder, USA, 2000.
[18] C.A. Ellis, S.J. Gibbs, G. Rein, Groupware: Some issues and experiences, Communications of the ACM 34 (1) (1991) 38–58.
[19] C.A. Ellis, K. Keddara, A workflow change is a workflow, in: W.M.P. van der Aalst, J. Desel, A. Oberweis (Eds.), Business Process Management: Models, Techniques, and Empirical Studies, Volume 1806 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2000, pp. 201–217.
[20] C.A. Ellis, K. Keddara, G. Rozenberg, Dynamic change within workflow systems, in: N. Comstock, C. Ellis, R. Kling, J. Mylopoulos, S. Kaplan (Eds.), Proceedings of the Conference on Organizational Computing Systems, Milpitas, California, ACM SIGOIS, ACM Press, New York, 1995, pp. 10–21.
[21] L. Fischer (Ed.), Workflow Handbook 2001, Workflow Management Coalition, Future Strategies, Lighthouse Point, Florida, 2001.
[22] M. Hammer, J. Champy, Reengineering the Corporation, Nicolas Brealey Publishing, London, 1993.
[23] E. Heinrich, H. Maurer, Active Documents: Concept, Implementation and Applications, Journal of Universal Computer Science 6 (12) (2000) 1197–1202.
[24] T. Herrmann, M. Hoffmann, K.U. Loser, K. Moysich, Semistructured models are surprisingly useful for user-centered design, in: G. De Michelis, A. Giboin, L. Karsenty, R. Dieng (Eds.), Designing Cooperative Systems (Coop 2000), IOS Press, Amsterdam, 2000, pp. 159–174.
[25] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, G. Zhou, Declarative workflows that support easy modification and dynamic browsing, in: G. Georgakopoulos, W. Prinz, A.L. Wolf (Eds.), Work Activities Coordination and Collaboration (WACC'99), ACM press, San Francisco, 1999, pp. 69–78.
[26] S. Jablonski, C. Bussler, Workflow Management: Modeling Concepts, Architecture, and Implementation, International Thomson Computer Press, London, UK, 1996.
[27] B. Karbe, N. Ramsperger, Support of cooperative work by electronic circulation folders, in: Conference on Office Information Systems, ACM Special Interest Group on Office Information SystemsACM SIGOIS, ACM Press, New York, 1990, pp. 109–117.

[28] S. Khoshanfian, M. Buckiewicz, Introduction to Groupware, Workflow, and Workgroup Computing, John Wiley and Sons, New York, 1995.
[29] M. Klein, C. Dellarocas, A. Bernstein (Eds.), Proceedings of the CSCW-98 Workshop Towards Adaptive Workflow Systems, Seattle, Washington, November 1998.
[30] M. Klein, C. Dellarocas, A. Bernstein (Eds.), Adaptive Workflow Systems, Volume 9 of Special issue of the journal of Computer Supported Cooperative Work, 2000.
[31] P. Lawrence (Ed.), Workflow Handbook 1997, Workflow Management Coalition, John Wiley and Sons, New York, 1997.
[32] F. Leymann, D. Roller, Production Workflow: Concepts and Techniques, Prentice-Hall PTR, Upper Saddle River, NJ, USA, 1999.
[33] London Bridge Group, Vectus Application Developer's Guide, London Bridge Group, Wellesbourne, Warwick, UK, 2001.
[34] London Bridge Group, Vectus Technical Architecture, London Bridge Group, Wellesbourne, Warwick, UK, 2001.
[35] D.C. Marinescu, Internet-Based Workflow Management: Towards a Semantic Web, Volume 40 of Wiley Series on Parallel and Distributed Computing, Wiley-Interscience, New York, 2002.
[36] M. Merz, B. Liberman, W. Lamersdorf, Using mobile agents to support interorganizational workflow-management, International Journal on Applied Artificial Intelligence 11 (6) (1997) 551–572.
[37] M. Merz, B. Liberman, W. Lamersdorf, Crossing organisational boundaries with mobile agents in electronic service markets, Integrated Computer-Aided Engineering 6 (2) (1999) 91–104.
[38] S.P. Nielsen, C. Easthope, P. Gosselink, K. Gutsze, J. Roele. Using Lotus Domino Workflow 2.0, Redbook SG24-5963-00, IBM, Poughkeepsie, USA, 2000.
[39] M. Reichert, P. Dadam, ADEPTflex: Supporting dynamic changes of workflow without loosing control, Journal of Intelligent Information Systems 10 (2) (1998) 93–129.
[40] H. Reijers, J. Rigter, W.M.P. van der Aalst, The case handling case, International Journal of Cooperative Information Systems 12 (3) (2003) 365–391.
[41] H. Reijers, K. Voorhoeve, On the optimal design of processes and information systems (in Dutch), Informatie 42 (December) (2000) 50–57.
[42] H. Schuschel, M. Weske, Integrated workflow planning and coordination, in: 14th International Conference on Database and Expert Systems Applicationvolume 2736 of Lecture Notes in Computer Science, Prague, Czech Republic, Springer-Verlag, Berlin, 2003, pp. 771–781.
[43] Software-Ley. COSA Activity Manager. Software-Ley GmbH, Pullheim, Germany, 2002.
[44] Staffware. Staffware Case Handler–White Paper. Staffware PLC, Berkshire, UK, 2000.
[45] M. Stonebraker, The integration of rule systems and database systems, TKDE 4 (5) (1992) 415–423.
[46] D.M. Strong, S.M. Miller, Exceptions and exception handling in computerized information processes, ACM Transactions on Information Systems 13 (2) (1995) 206–233.
[47] Tibco, TIB/InConcert Process Designer User's Guide, Tibco Software Inc., Palo Alto, CA, USA, 2000.
[48] E. Verharen, F. Dignum, S. Bos, Implementation of a cooperative agent architecture based on the language-action perspective, in: M. Singh, A. Rao, M. Wooldridge (Eds.), Agent Theories, Architectures, and Languages, Volume 1365 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1998, pp. 31–44.
[49] E.M. Verharen, F. Dignum, S. Bos, Implementation of a cooperative agent architecture based on the language-action perspective, in: Intelligent Agentsvolume 1365 of Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 1998, pp. 31–44.
[50] Peter Wegner, Why interaction is more powerful than algorithms, Communications of the ACM 40 (5) (1997) 80–91.
[51] M. Weske, Formal Foundation, Conceptual Design, and Prototypical Implementation of Workflow Management Systems, Habilitation's thesis, University of Münster, Germany, 2000.
[52] M. Weske, Formal foundation and conceptual design of dynamic adaptations in a workflow management system, in: R. Sprague (Ed.), Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-34), IEEE Computer Society Press, Los Alamitos, California, 2001.

Wil van der Aalst is a full professor of Information Systems and head of the Information Systems department of the Faculty of Technology Management at Eindhoven University of Technology. Currently he is also an adjunct professor at Queensland University of Technology (QUT) working within the Centre for Information Technology Innovation (CITI). His research interests include information systems, simulation, process mining, Petri nets, process models, workflow management systems, verification techniques, enterprise resource planning systems, computer supported cooperative work, and interorganizational business processes.

Mathias Weske is a professor of computer science and chair of the business process technology research group at Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam. His research interests include workflow management, business process management, software architectures for process-oriented information systems, service oriented computing, and software product lines. He is a member and Vice Chair of the executive committee of GI SIG EMISA (German Computer Science Society Special Interest Group on Development Methods for Information Systems and their Application), and a member of IEEE and ACM.

Dolf Grünbauer is a senior software engineer of FLOW*er*, a product developed by Pallas Athena. He has a Master of Science degree in Applied Mathematics at the University of Twente. Since 1990 he has been working on architecture, design, theoretical foundation and implementation of workflow—and case handling systems for different companies across the software industry.