Enacting Interorganizational Workflows Using Nets in Nets

Wil van der Aalst[1], Daniel Moldt, Rüdiger Valk, and Frank Wienberg[2]

[1] Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, `wsinwa@win.tue.nl`

[2] University of Hamburg, Department for Computer Science, Vogt-Kölln-Str. 30, D-22527 Hamburg, {`moldt, valk, wienberg`}`@informatik.uni-hamburg.de`

Abstract. The primary task of a workflow management system is to enact case-driven business processes by joining several perspectives, including the control-flow perspective, the resource perspective, the data perspective, the task perspective, and the operation perspective. In this paper, we propose *reference nets*, a particular class of Petri nets where the tokens can be references to other Petri nets, as a technique for clearly specifying each of the perspectives *and* the relation between these perspectives. The "nets in nets" paradigm offered by reference nets also allows to model *mobility* of a business object like a workflow instance, a resource, a data element, a task, or an operation. Therefore, reference nets are particularly suitable for specifying and enacting interorganizational workflows. To hide aspects only relevant for a single organization, we use advanced inheritance concepts to facilitate the exchange of business objects across organizational boundaries. To show the applicability of the approach, we have implemented a simple workflow engine using *Renew*: a Java-based interpreter of reference nets.

Keywords: Inheritance, Nets in nets, Reference nets, Workflow, Workflow management system

1 Introduction

This paper proposes the use of *reference nets* [KW99] to tackle the following two problems. The first problem is the unclear mixture of perspectives in the current generation of workflow management systems making workflow specifications incomplete and difficult to interpret. The second problem is the absence of tools to describe and enact the mobility of business objects required for interorganizational workflows.

To address the first problem, we introduce some of the basic workflow terms using the following five perspectives: (1) *control-flow* (or routing) perspective, (2) *resource* (or organization) perspective, (3) *data* (or information) perspective, (4) *task* (or function) perspective, and (5) *operation* (or application) perspective. Since the main focus of this paper is not to model any of these perspectives in detail, but to model their relations, we only give a brief description of each perspective. Refer to [Aal98] and (for a similar approach) to [JB96] for further details. In the control-flow perspective, *workflow process definitions* (workflow schemas) are defined to specify which *tasks* need to be executed and in what order. Workflow process definitions are instantiated for specific *cases*. Since a case is an instantiation of a process definition, it corresponds to the execution of concrete work according to the specified routing. In the *resource* perspective, the organizational structure and the population are specified. The data perspective deals with *control* and *production data*. Control data are data introduced solely for workflow management purposes, e.g., variables introduced for the routing of cases. Production data are information objects (e.g., documents, forms, and tables) whose existence does not depend on workflow management. The task perspective describes the elementary operations performed by resources while executing a task for a specific case. In the operational perspective, the relation between operations, data and applications is given. Typically, operations create, read, or modify control and production data in the information perspective. (In this paper, we will focus on control data.) Most operations are (partially) implemented by applications. A *workflow definition* is the specification of a workflow covering all aspects and linking the five perspectives together (see Figure 1). Cases are instances of a workflow definition and are handled accordingly. A *workflow management system* aims at supporting the five perspectives shown in Figure 1. The build-time part of the workflow management system allows for the specification of these perspectives. The run-time part of the workflow management system takes care of the actual enactment.

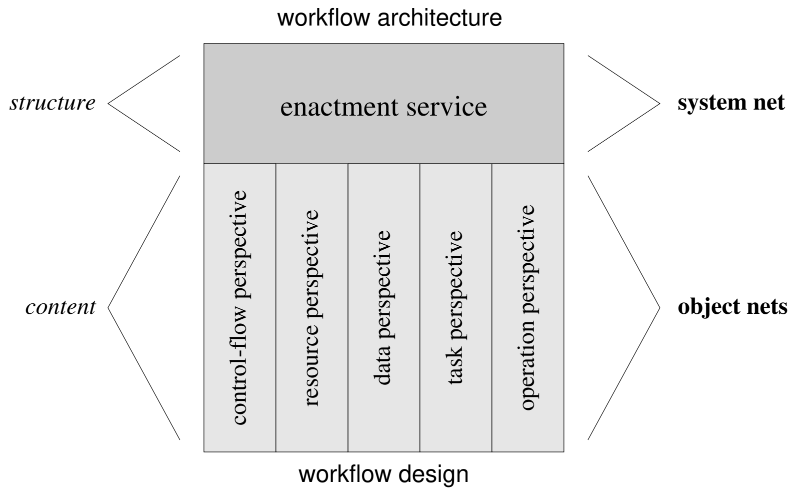

Fig. 1. The five perspectives considered in this paper.

Today's workflow management systems have problems specifying the relations between these perspectives. Most commercial workflow management systems only allow for the explicit specification of the control-flow perspective and the resource perspectives. All the other perspectives are modeled implicitly in the control-flow perspective. For example, data elements, operations, and tasks are defined *inside* the workflow process definition. The fact that the control-flow perspective dominates all other perspectives restricts the reuse of parts of the workflow definition and limits the extensibility of the workflow management system with additional perspectives. Moreover, since all perspectives are intertwined, it is impossible to exchange cases, resources, data, tasks, or operations between workflow enactment services in different organizations. This brings us to the second problem addressed in this paper.

E-commerce, in its earliest incarnation mainly driven by *Electronic Data Interchange* (EDI), has traditionally been used by larger corporations to share and exchange information between business partners and suppliers. However, with the explosive growth of the Internet in the last couple of years and emerging technologies such as Java and XML, electronic commerce is now able to offer solutions for a much broader range of business processes than EDI previously addressed. Moreover, E-commerce technologies can also be used inside companies. As a result of these developments, *interorganizational workflows* [Aal99] are becoming more important. One can think of workflows crossing organizational boundaries between corporations in an electronic-commerce setting (business-to-business E-commerce) but also of workflows involving multiple business units within one corporation (i.e., *intra*organizational workflows). Today's commercial workflow management systems use a centralized enactment service, and therefore, have problems dealing with interorganizational workflows. This centralized view is reflected in both the build-time and the run-time part of the workflow management system. Therefore, fundamentally new concepts are needed to support interorganizational workflows in a comprehensive manner.

In this paper, we propose *reference nets* as a partial solution for the two problems identified. Reference nets are a class of Petri nets using the "nets in nets" paradigm [Val87]. Using this paradigm tokens in the Petri net are represented by Petri nets. Many variants of such an approach have been proposed [Lak95,Hol95,Val87]. In the notion of *object-nets* [Val98], tokens of a so-called system-net correspond to marked Petri nets on a lower level, called object-nets. Since the object-nets actually reside in the system-net, we call this the value-semantics approach. Reference nets use another approach: The object-nets do not actually reside in the system-net, but tokens are references to object-nets. This means that multiple tokens can reference the same object-net. Therefore, in analogy to programming language, we use the term reference semantics. In this paper, we want to illustrate the enactment service based on nets in nets by an actually executable model. We were forced to use nets according to the reference semantics for pragmatic reasons: Renew (The Reference Net Workshop, [KW99]) is to our knowledge the only tool supporting execution of any kind of nets in nets, and it uses reference semantics.

We model the five perspectives, i.e., the control flow, resource, data, task, and operation perspective, in terms of reference nets. An instance of each perspective corresponds to one marked object-net. The system-net joins all perspectives and can be seen as the enactment service of a workflow management system. Since every aspect is modeled in a separate object-net, it is not necessary to intertwine all aspects. Moreover, the system-net is generic, i.e., independent of actual workflows and organization. One can think of the system-net as an *architectural model* and the object-nets as the actual *content*. The workflow designer only creates object-nets. The system-net is given by the desired characteristics of the workflow management system. This division between structure (system-net) and content (object-nets) has some interesting features. First of all, the same object-nets can be used in different system nets representing different architectures. Second, a system-net can be used to enact arbitrary workflows satisfying some minimal requirements. The clear division between structure and content makes the relations between the perspectives explicit and subject to discussion, whereas in most workflow management systems these issues remain hidden. (Mobile [JB96] is one of the rare systems clearly separating the different perspectives.) Another important feature of the "nets in nets" approach is the ability to describe mobility in a direct and transparent manner. By simply moving a token corresponding to an object-net from one place to another, we can model the exchange of cases, resources, data elements, tasks, and operations.

For interorganizational workflows it is not realistic to assume that all organizations involved use the same workflow enactment service. Moreover, the perspectives are typically modeled in a different way. For example, one organization uses more tasks to handle a case than another organization. Another example is the modeling of resources: The rules with respect to the allocation of resources may vary from one organization to another. For this purpose, we use the inheritance notions introduced in [AB97,Bas98,BA99]. If one object-net is a subclass of another object-net, then the transfer rules presented in [AB99] can be used to map the state of the superclass onto the subclass and vice versa. This means that if there is a subclass/superclass relation between perspectives in different organizations, then there is no problem to migrate cases, resources, data elements, tasks, and operations, i.e., mobility is guaranteed.

To illustrate our approach based on reference nets and advanced inheritance notions, we have developed a prototype enactment service using Renew.

In the remainder of this paper, we first introduce reference nets and the Renew tool. Then we model the five perspectives using object-nets. In Section 4 we link these perspectives together using a generic system-net. Then, we focus on interorganizational workflows and the role of inheritance to facilitate the migration of object-nets. Finally, the strengths and weaknesses of the approach proposed are evaluated.

2 Reference nets: Formalism and Tool

The *reference net* formalism defines a special class of high-level Petri nets that uses Java as an inscription language and extends Petri nets with dynamic net instances, net references, and dynamic transition synchronization through synchronous channels. Reference nets consist of *places* (graphically represented by ellipses), *transitions* (boxes), and *arcs* (lines with arrow tips). There are, in essence, three types of arcs. Firstly, ordinary *input* or *output arcs* that come with a single arrow head. These behave just like in ordinary Petri nets, removing or depositing tokens at a place. Secondly, there are *reserve arcs*, which are simply a shorthand notation for one input and one output arc. Effectively, these arcs reserve a token during the firing of a transition. Thirdly, there are *test arcs*, which have no arrowheads at all. A single token may be accessed, i.e., tested, by several test arcs at once.

Each place or transition may be assigned a *name*, displayed in bold type.

Every net element can carry semantic inscriptions. Places can have an arbitrary number of *initialization expressions*. The initialization expressions are evaluated and the resulting values serve as initial markings of the places. By default, a place is initially unmarked. Arcs can have an optional *arc inscription*. When a transition fires, its arc expressions are evaluated and tokens are moved according to the result. Transitions can be equipped with a variety of inscriptions. *Expression inscriptions* are ordinary expression that are evaluated while the net simulator searches for a binding of the transition. The inscriptions are Java expressions. The result of this evaluation is discarded, but in such expressions you can use the equality operator `=` to influence the binding of variables that are used elsewhere. *Guard inscriptions* are expressions that are prefixed with the reserved word `guard`. A transition may only fire if all of its guard inscriptions evaluate to `true`. By this we cover the basic colored Petri net formalism.

There are two kinds of tokens: valued tokens and tokens which correspond to a reference. By default, an arc will transport a *black token*, denoted by `[]`. But if you add an *arc inscription* to an arc, that inscription will be evaluated and the result will determine which kind of token is moved.

Variables are bound to one single value during the firing of a transition. However, during the next firing of the same transition, the variables may be bound to completely different values. This is quite similar to the way variables are used in logical programming, e.g., in Prolog.

The inscription language of reference nets has been extended to include *tuples*. A tuple is denoted by a comma-separated list of expressions that is enclosed in square brackets. Tuples are useful for storing a whole group of related values inside a single token and hence in a single place.

Additionally there are *creation inscriptions* that deal with the creation of net instances, and *synchronous channels*. A net is specified as a static structure. However, an instance of the net has a marking that can change over time. Whenever a simulation is started, a new *net instance* is created. Every net has to be given a *name*. New net instances are created by transitions that carry *creation inscriptions*, which consist of a variable name, a colon (`:`), the reserved word `new`, and the name of the net.

A net does not disappear simply because it is no longer referenced. On the other hand, if a net instance is no longer referenced and none of its transition instances can possibly become enabled, then it is subject to garbage collection.

Net instances need some means of communication. We chose to use *synchronous channels*, which were fist considered for colored Petri nets by Christensen and Damgaard Hansen in [CH92]. They synchronize two transitions which both fire atomically at the same time. Both transitions must agree on the name of the channel and on a set of parameters before they can synchronize.

Here we generalize this concept by allowing transitions in different net instances to synchronize. In association with classical object-oriented languages we require that the initiator of a synchronization owns a reference ("knows") the other net instance.

The initiating transition must have a special inscription, the so-called *downlink*. A downlink makes a request at a designated subordinate net. A downlink consists of an expression that must evaluate to a net reference (usually a variable), a colon (`:`), the name of the channel, and an optional list of arguments.

On the other side, the transition must be inscribed with a so-called *uplink*. An uplink serves requests for everyone. Therefore the expression that designates the other net instance is missing for uplinks.

In reference nets, `this` denotes the net instance in which a transition fires.

Generally, transitions with an uplink cannot fire without being requested explicitly by another transition with a matching downlink. It is allowed that a transition has multiple downlinks. It is also allowed that a transition has both an uplink and downlinks.

Channels can also take a list of parameters. Although there is a direction of invocation, this direction need not coincide with the direction of information transfer. Indeed it is possible that a single synchronization transfers information in both directions.

A tool for specifying and executing reference nets called Renew (Reference Net Workshop) is used here to build the executable models of this contribution. Renew itself is a Petri-net-based software package developed by members of the Computer Science department of the University of Hamburg. It is implemented in Java and is freely available [KW99]. It offers an intuitive GUI for building net models and viewing simulation runs. Both reference nets and their supporting tool Renew are based on the programming language Java. To be able to use them to their full capacity, some knowledge of Java is required.

The main strength of Renew lies in its openness and versatility. It is important to notice that the use of Java (or a similar language) is the prerequisite to allow for the portability.

3 Modeling perspectives as object-nets

Each of the perspectives introduced in the following can be represented independent of the other perspectives, but interfaces have to be provided to have reasonable models. These interfaces to the other perspectives have to be considered carefully. Using reference nets object-oriented principles can be applied to support the separation of concerns. In this section we will describe each perspective separately by reference nets with the interpretation that these nets are object-nets. The integration within the system-net follows in the next Section 4.

3.1 Control-flow perspective

Workflows and their instances can be modeled using sound workflow nets as defined in [Aal98]. Due to the use of reference nets this definition is adopted here. A *workflow net* (WF-net) is a reference net with one source transition and one sink transition. The source transition has a synchronous channel named `new()` and the sink transition has one named `done()`. Every node of the net is on a path from the source to the sink. Except for synchronous channels the net inscriptions are not included in this definition.

Not every workflow net is acceptable, e.g., the workflow net should not deadlock. Therefore, the *soundness property* defined in [Aal98] is used. For sound behavior of workflow nets it is necessary that after instantiation (i.e., the firing of the source transition `new()`) proper termination is guaranteed: From all possible reachable states termination (i.e., the firing of the sink transition `done()`) must be possible and after termination no part is allowed to be active anymore and all places have to be empty or may only contain references to inactive parts, e.g., to the initial data of the net instance. When a workflow instance is instantiated, the source transition has to synchronize with the environment to start this case. The instance will be deleted when no reference to the workflow instance exists and the synchronous channel `done` is executed.

Figure 2 shows an example of a workflow net. Here a fork and a join for the

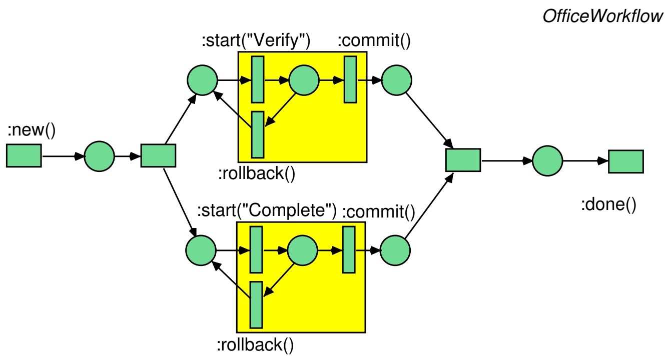

Fig. 2. An example of a workflow net modeling tasks in an office.

parallel execution of the example tasks `Verify` and `Complete` are modeled (also called AND-Split and AND-Join, respectively). Each task has a start and two ends in this particular model. The execution of a task can fail and hence at the end it has to be determined whether a commit or a rollback has to occur. By simply removing the rollback opportunity the structure becomes simpler. For each task the possibility of a failure can be modeled within the workflow, hence this kind of information belongs to the workflow.

The communication to other perspectives is done by the communication with the tasks which are triggered by the workflow. For this the synchronous channels are used. The names used in Figure 2 are `new` for the initialization of the workflow, `start(taskname)` for the start of the task with the name `taskname`, `commit` for the successful termination of the task, `rollback` for the unsuccessful termination of the task, and `done` for the termination of the workflow.

The fork and join tasks are "silent" actions. They occur without interaction with the environment, i.e. the system-net. If the system model should also cover the control of this kind of actions, then the appropriate channels can easily be added.

3.2 Resource perspective

Resources can have an arbitrary behavior which is reflected in their models. However, in this contribution we use a very simple model, as can be seen in Figure 3 which presents a trivial behavior. In Figure 3 the initialization is done by

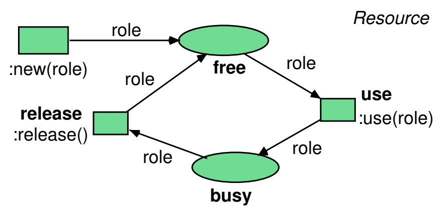

Fig. 3. An example of a Resource net.

the `new(role)` channel which receives the parameter `role`. This allows providing exactly one role for a resource. This role is used in the `use` channel to determine the role in which a resource can be used. In our example exactly one role is present. However, more roles can easily be modeled. The `release` channel needs no parameter hence the active role is determined by the `use` channel. The specific behavior of the resource can of course be extended without any problems.

3.3 Data perspective

The data perspective covers only the control data, i.e., only the data relevant for routing purposes. All other data is controlled by the application or, in our terms used here, the operations.

The net in Figure 4 shows a very simple data model. Via the synchronous channel `create(name,value)` the environment can install data with the name `name` and with the content `value`. Via the `read` channel the environment can inspect the current value of the data. The `update` channel allows for the modification of the data. Using `remove` the data can be removed. In principle, the

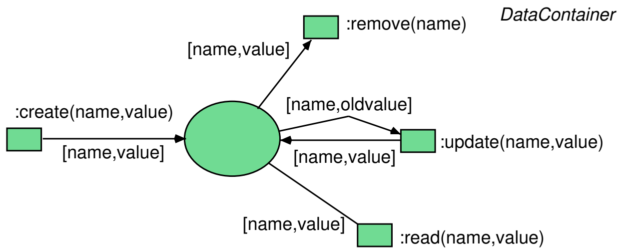

Fig. 4. An example of a data net.

simple structure can become arbitrarily complex. However, here the structure can be kept simple, since data instances are associated to specific workflow instances.

From the data there is no link to other perspectives. Note that production data could be modeled similarly. However, for reasons of simplicity we abstract from these data.

3.4 Task perspective

Tasks are a central issue concerning the execution of an workflow. They constitute the specific actions within a workflow instance. Structurally they are similar to workflow nets. Therefore, we assume that they have all the properties of a sound workflow net mentioned earlier. However, while they have one `start` transition there may be a `fail` transition in addition to the `end` transition.

In Figure 5 and Figure 6 two examples are given. In Figure 5 the `start` tran-

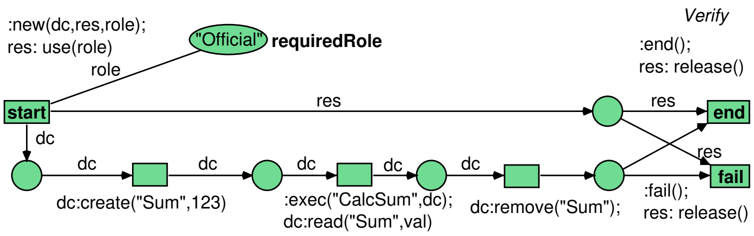

Fig. 5. An example of a task net (Verify).

sition has two synchronous channels. One for the creation of the task instance (`new(dc,res,role)`) and one for the handling of roles which are allowed for this particular type of task. The parameters are related to the data of the workflow (`dc`), to the resource required for this task instance (`res`), and to the role which is determined within the task (`role`). In this case the only permissible role is `Official`. The environment has to provide officials which are used to execute the task. Within this example the resources are not related to the execution. If necessary, this could be done. The data reference is used to access the relevant data via `create` and `read`. While here only some dummy actions are performed, arbitrary complex ones can be modeled. The environment has to provide some means to synchronize with the `exec("CalcSum",dc)` and the `dc:read("Sum",val)`. This is an example for the execution of an operation `CalcSum` to which the data is passed in form of `dc`. Further actions like the read on some data or even some communication with the resource is possible. The latter is not modeled here.

In general the operations can be manual, like `check_container` in a container terminal in the harbor of Hamburg, automatic, like `calculate_the_tax_sum` in a sale department, or it can be a mixture of both. The first is typically performed by humans while the second operation is performed by specific application programs.

In Figure 6 a different kind of task is modeled. The difference to `Verify` is

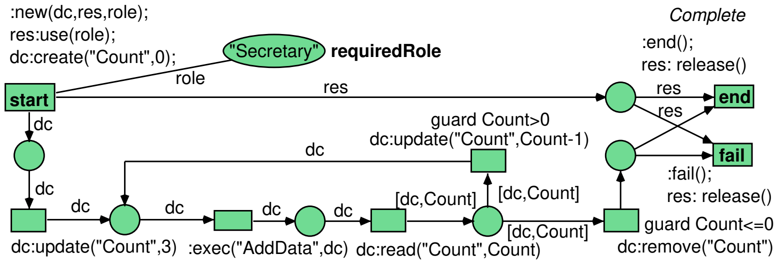

Fig. 6. An example of a task net (Complete).

that the initialization of data is done with the `start` transition, the required role is `Secretary`, and that there is a loop where the operation `AddData` is called at least once. The value put into the data here is `3`. `AddData` is executed and then the value is read from the data. Guards are used to determine the control flow with this task. Finally the task terminates either successfully or it fails.

Interfaces exist to the resources, the data, and the operations.

3.5 Operation perspective

The operations are kept very simple. They represent the concrete items that are executed by the task. Figure 7 shows a generic model of an operation which can be given a name during instantiation (see `new`) that is used for performing the operation (see `perform`).

The operation can only be performed when it is in the internal state `available`. Due to the autonomous behavior of object-nets, the operation can become unavailable at any time without control of the system-net. Operations are considered to be atomic. Therefore a single transition with an associated synchronous channel is necessary for each interaction with the environment. If more complex structures are needed, this is a hint to use tasks or even workflows to model the content of this "operation".

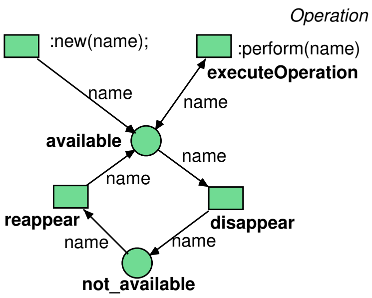

Fig. 7. An example of an operation net.

In this section we showed that the perspectives can be modeled in terms of object-nets and that each object net corresponds to a sound workflow net. This results in decoupled models which still need to be integrated. In the next section a system-net is given to integrate all models to represent the complete architecture of one workflow engine.

4 Linking the perspectives to enact workflows within one organization

In this section we discuss the integration of the introduced perspectives. A system-net is given that represents the general architecture of a workflow engine for one organizational unit. Between different organizational units this engine may vary. Here only one version is presented. Figure 12 shows the relationship between the different sites. The migration of workflows, data, resources etc. between these sites is discussed in Section 5.

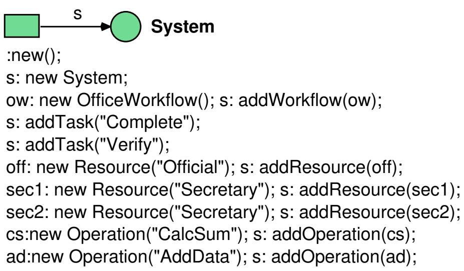

Fig. 8. An example of creating a concrete instance of the system-net.

The model in Figure 9 represents the system-net of one workflow engine. In the upper right corner is a declaration that imports code from the Renew library. Figure 8 represents the corresponding initializations needed for a concrete simulation. An instance of a system net is created and equipped with the

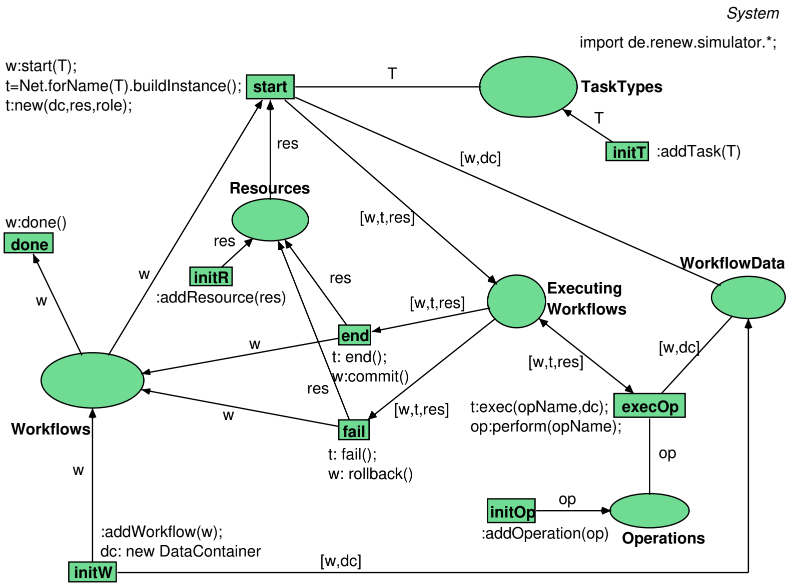

Fig. 9. An example of a system-net.

example workflow, tasks, resources, and operations. The transition inscription in Figure 8 which is responsible for performing these initializations starts with the uplink `new`. This uplink is automatically invoked at the beginning of the execution. Then, a workflow instance and its corresponding data are created and initialized via the synchronous channel `addWorkflow`, which synchronizes with the `initW` transition of the system net in Figure 9. Synchronously, the two example tasks `Verify` and `Complete` and declared using the `addTask`-channel, so that the system net can check if tasks specified by workflows do really exist. Then three resources are added via the same kind of mechanisms (see `initR` and `addResource(res)`): One `Official` and two `Secretarys`. Finally two operations are created, `CalcSum` and `AddData`. It is important to note that all initializations are invoked from the environment via a call of the appropriate synchronous channels, making the system net entirely independent of concrete workflows, tasks, resources, and operations.

The structure of the engine is very simple, but also very generic. Workflows can be started (see `start`). This requires a workflow, a resource, and a task (template reference) and results in an executing workflow. The synchronous channels describe how the possible bindings are restricted and how the relation between the system-net and the object-net are defined. From the place of the executing workflows the transition `end` for successful termination and the transition `fail` for a failure of the case execution are used. In both cases resources and workflows are restored in their "original" places.

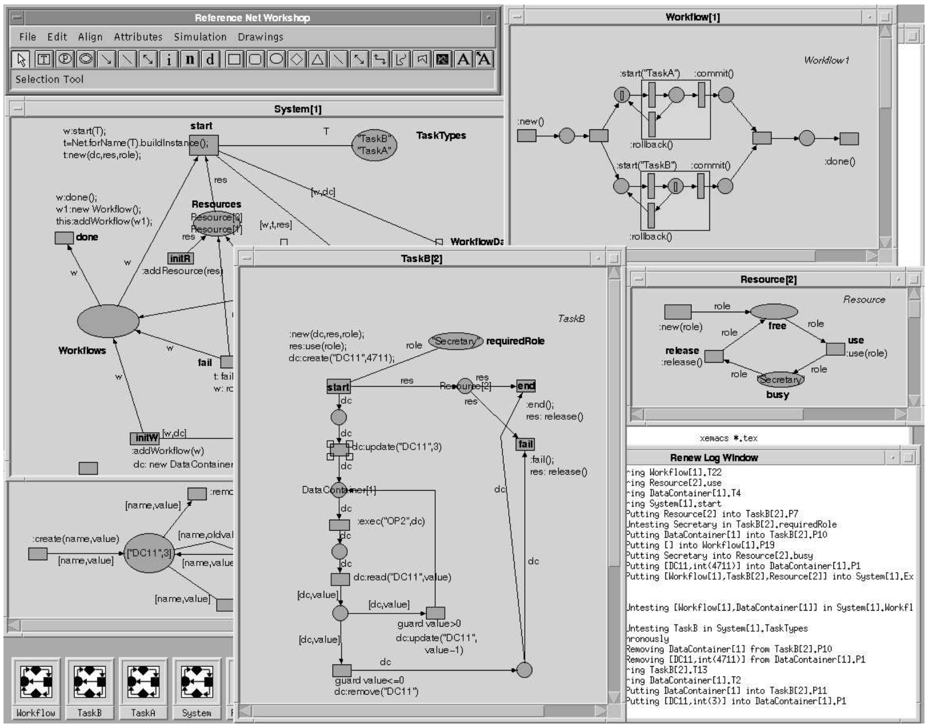

Fig. 10. Screenshot of the Renew tool at work.

5 Interorganizational workflow and inheritance

In this section, we will show that the "nets in nets" paradigm also enables *mobility* in the various perspectives: Moving a business object (e.g., workflow instance, a resource, a data element, a task, or an operation) from one system to another can be modeled in a uniform manner. Mobility only makes sense if there is some degree of freedom, i.e., it should be able to allow some variations with respect to the perspectives identified. For this purpose, we use inheritance.

5.1 Inheritance of dynamic behavior

Inheritance is one of the cornerstones of object-oriented programming and object-oriented design. The basic idea of inheritance is to provide mechanisms which allow for constructing subclasses that inherit certain properties of a given superclass. In our case a class corresponds to an object-net class (i.e., the definition of a net used to describe one of the five perspectives) and objects correspond to object-net instances. Recall that each object-net class corresponds to a sound workflow net, i.e., an object life-cycle without deadlocks, livelocks, and other anomalies. The work presented in [AB97,Bas98] deals with inheritance of dynamic behavior such object life-cycles. In particular, we will use of the notion of *projection inheritance* to tackle the problems indicated in this paper. Note that projection inheritance is just one of the four inheritance notions presented in [AB97,Bas98].

The notion of *projection inheritance* is based on abstraction. Let x and y be two object-net classes each represented by a sound workflow net. *If it is not possible to distinguish x and y when arbitrary tasks of x are executed, but when only the effects of tasks that are also present in y are considered, then x is a subclass of y with respect to projection inheritance.* For distinguishing x and y under projection inheritance we only consider the tasks present in both nets (i.e., in y). All other tasks in x are renamed to τ. One can think of these tasks as silent, internal, or not observable. Since branching bisimulation (see [Bas98] for detailed information and literature pointers) is used as an equivalence notion, we abstract from transitions with a τ label, i.e., for deciding whether x is a subclass of y only the tasks with a label different from τ are considered. The behavior with respect to these tasks is called the observable behavior. Added tasks (i.e., tasks present in x but not in y) can be executed but are not observable by the outside world, i.e., projection inheritance conforms to hiding or abstracting from tasks new in x.

Figure 11 shows five workflow processes modeled in terms of workflow nets. Workflow process (A) consists of three sequential tasks: *register*, *handle*, and *archive*. Each of the other workflow processes extends this process with one additional task: *check*. Workflows (B), (D), and (E) are subclasses of (A) with respect to projection inheritance. Workflow process (B) is a subclass of workflow process (A) with respect to projection inheritance: If task *check* is abstracted from, then the two processes behave equivalently (i.e., are branching bisimilar). Workflow process (C) is not a subclass with respect to projection inheritance: Hiding task *check* introduces the possibility to skip task *handle* and thus change the actual behavior. Workflow process (D) is a subclass of workflow process (A) with respect to projection inheritance: Hiding this task results in two equivalent processes. Workflow process (E) is a subclass of workflow process (A) with respect to projection inheritance: The detour via task *check* can be hidden thus yielding an observable behavior identical to (A).

In [AB97,Bas98] we proposed a number of *inheritance-preserving transformation rules*. These rules correspond to frequently used design constructs and preserve one or more of the four inheritance notions. A detailed description of these rules is beyond the scope of this paper, but given in [AB97,Bas98]. Therefore, we just give an informal description of the three rules that preserve projection inheritance: PP, PJ, and PJ3. Details and subtle requirements are omitted to simplify the presentation of the main ideas.

- Tranformation rule *PP* can be used to add loops: The extension (i.e., the added subnet) takes a token from a place in the original net but returns the token after a while.
- Inheritance-preserving transformation rule *PJ* inserts new tasks in-between two tasks in the original net. The added subnet may have any structure

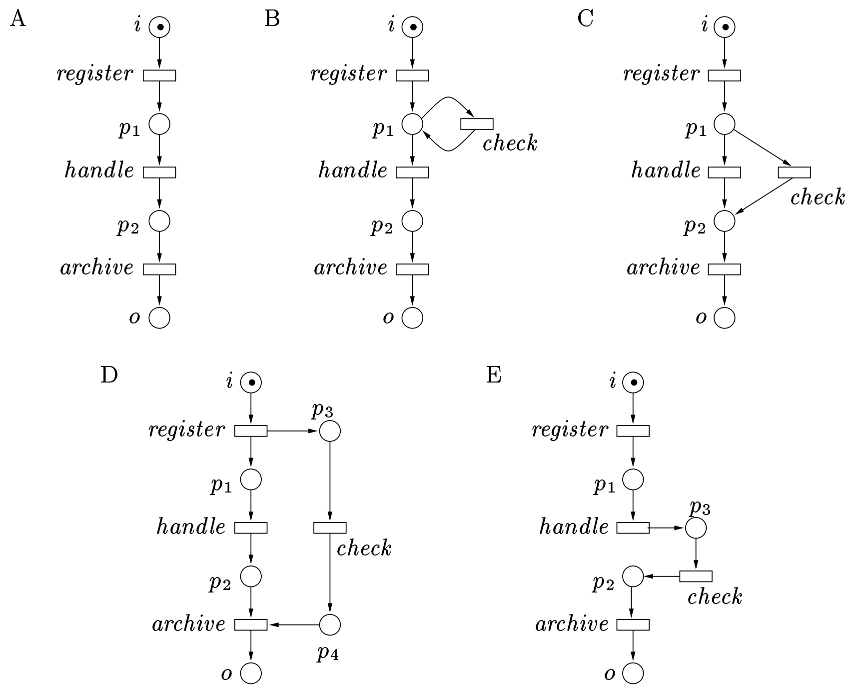

Fig. 11. Five routing diagrams describing variants of a simple workflow process.

as long as it is guaranteed that once the subnet is activated eventually the control is returned to the original net.

– Transformation rule *PJ3* also preserves projection inheritance and can be used to add parallel behavior. (The rule is named PJ3 for historical reasons.) The added subnet may use arbitrary routing constructs as long as some straightforward properties are satisfied.

The workflow nets shown in Figure 11 can be used to illustrate the four rules. The rules PP, PJ, and PJ3 can be applied to construct the workflows (B), (E), and (D) from workflow (A) respectively. The three rules correspond to design constructs that are often used in practice, namely iteration, sequential composition, and parallel composition. If the designer sticks to these rules, inheritance is guaranteed. Moreover, if the designer sticks to the inheritance-preserving transformation rules, then it is possible to guarantee instant transfers, i.e., it is possible to map states from a subclass to a superclass and vice versa. In [AB99] these *transfer rules* are defined formally. Suppose that x is a subclass of y constructed using the rules PP, PJ, and PJ3. For any state in workflow process y it is possible to transfer a case to x such that the transfer is instantaneous (i.e., no postponements needed) and does not introduce syntactic errors (e.g., deadlocks, livelocks, and improper termination) nor semantic errors (e.g., the double execution of tasks or unnecessary skipping of tasks). Moreover, it is also possible to transfer cases from subclass x to superclass y without any problems.

5.2 Putting inheritance to work

In the introduction we mentioned two benefits of the use of reference nets for (interorganizational) workflows: (1) clear separation of the perspectives and systems architecture, and (2) mobility. i.e., the ability to move objects corresponding to the various perspectives. The first benefit was already demonstrated in the first part of the paper. In the remainder we will focus on mobility. However, we first show that inheritance also allows for the specification of constraints with respect to the object-net classes used for the various perspectives.

In Section 3, it was stated that the object-net classes used for the control-flow perspective must be sound workflow nets similar to those defined in [Aal98]. This can be checked using the verification tool Woflan [VA]. The object-net class used to model the resource perspective should be a subclass *under projection inheritance* of the object-net class shown in Figure 3. This implies that the minimal life-cycle model for a resource comprises a *release* method and a *use* method. These two methods need to be executed alternatingly. It is allowed to add new methods that are executed in-between the existing methods (rule PJ), in parallel (rule PJ3), or using the loop construct (rule PP). Similar remarks hold for the other perspectives. The object-net class used to model the data perspective should be a subclass (under projection inheritance) of the object-net class shown in Figure 4. Additional methods could be introduced to model more details about the management of data. The object-net class which brings into play the task perspective should be an object life-cycle (i.e., a sound workflow net [Aal98]) starting with an operation to allocate a resource and an "end operation" and/or a "fail operation" (both releasing the resource allocated). Finally, the object-net class which specifies the operation perspective should be a subclass under projection inheritance of the object-net class shown in Figure 7.

Inheritance plays an important role in enabling mobility between different enactment services. The "nets in nets" paradigm allows for the transfer of object-nets from one system-net to another system net. Consider for example the schematic representation shown in Figure 12. This reference net connects three system nets via so-called *transfer transitions*. Each of the three enactment services (A, B, and C) corresponds to a system-net such as the one shown in Figure 9. Each transfer transition transfers one or more object-nets from one enactment service to another. Consider for example transition *Transfer_case_AB* which transfers a workflow instance (i.e., a case) and the associated data from enactment service A to enactment service B. The transitions *Transfer_case_BC* and *Transfer_case_CA* also move workflow instances and the associated data from one location to another. Moving a case with its associated data is easier than just moving the case. If a case and its associated data reside at different locations, there has to be a mechanism to get to the workflow related data. Note that middleware such as CORBA can be used to simplify the access to remote data. Figure 12 also shows two other transfer transitions: *Transfer_resource_BC* and

Transfer_resource_CB. These transitions move resources (in the form of object-nets) from enactment service B to enactment service C and vice versa.

Figure 12 illustrates the modeling power of reference nets: In one model, it is possible to deal with both the actual workflows represented by the object-nets and architectural considerations such as the transfer and distribution of cases, data, resources, operations, and tasks. Note that related work on workflow inheritance [Bus99] does not consider these mobility aspects.

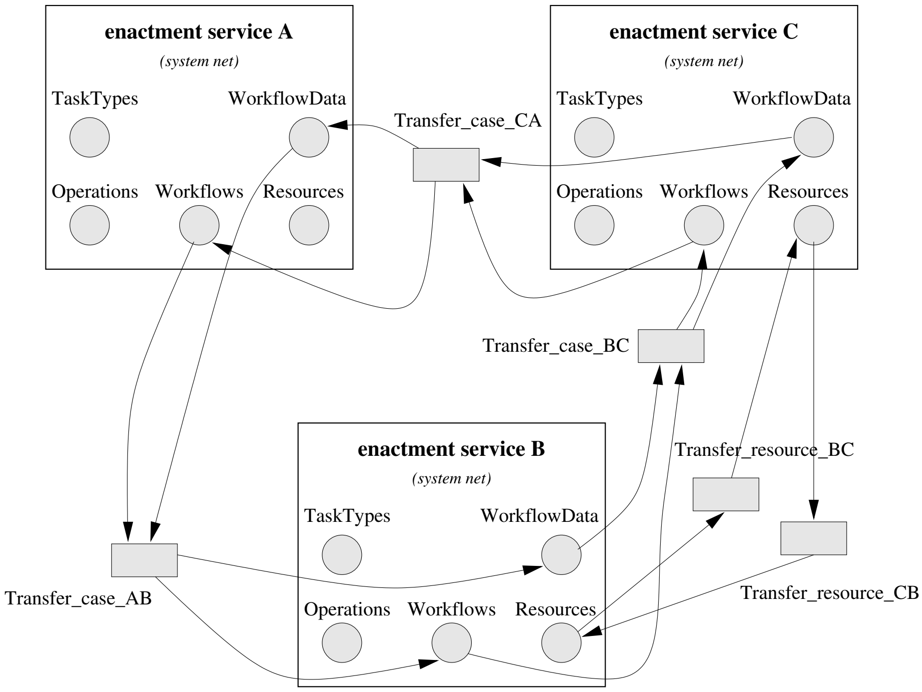

Fig. 12. Exchanging object-net instances to distribute work and resources.

The constellation of system-nets and transfer transitions subsumes a *transfer policy*. The transfer policy should provide answers to the following two questions:

– *When* to transfer?
  There are many reasons for a transfer. Consider for example the transfer of a case (i.e., workflow instance). The case can be transferred because (1) there are no resources qualified to execute a given task, (2) another department is responsible, (3) to balance the workload, (4) the customer the case refers to moved to another city, or (5) the associated goods are transported to another country. Resources can be transferred for capacity balancing or other organizational reasons. Data, tasks, and operations can also be transferred or replicated for various reasons.

– *What* to transfer?
  If a case is moved, it may be good to move the associated data (see Figure 12). However, it is also possible to centralize data and only distribute the control flow. The perspectives can be bound tightly together or there can be a very loose connection between the perspectives. For example, data, operations, resources, and tasks related to a case may reside at different locations.

It is far from trivial to devise a suitable transfer policy. The policy should be based on architectural considerations of both an organizational and technical nature. We will not discuss this in any detail. However, it should be clear by looking at examples such as the one shown in Figure 12 that reference nets allow for the modeling of these mobility aspects.

Concerning the relation to inheritance, it is essential for interorganizational workflows that there can be local variations, e.g., one department uses a slightly different control flow or the life-cycle of a resource has some local particularities (e.g., switching between day-shifts and night-shifts). To allow for these variations, we use the inheritance concepts defined earlier. If an object-net instance is moved from one enactment service to another, then the corresponding two object-net classes should have a subclass/superclass relationship. This way it is possible to map the object-net instance onto the new object-net class using the transfer rules defined in [AB99]. Recall that the inheritance-preserving transformation rules PP, PJ, and PJ3 are augmented with transfer rules for moving tokens from the subclass to the superclass and vice versa. The same rules can be used to map an object-net instance onto a superclass or a subclass. The transfer rules to move an object-net instance to a subclass are: r_{PP}, r_{PJ}, , $r_{PJ3,C}$ and $r_{PJ3,P}$ ([AB99]). Transfer rules r_{PP}, and r_{PJ} are rather trivial because additional behavior (i.e., alternative branches or parts inserted in-between existing parts) is introduced without eliminating existing states. The transfer rule corresponding to transformation rule PJ3 is more complex because PJ3 adds parallel behavior rather than additional behavior. When adding parallel behavior, it may be necessary to mark places in the newly added parts. If this is the case, there is a choice to put the tokens in the beginning of the parallel part (conservative approach $r_{PJ3,C}$) or to put the tokens at the end of the parallel part (progressive approach $r_{PJ3,P}$). This choice depends of the desired policy. The transfer rules to move an object-net instance to a superclass are: r_{PP}^{-1}, r_{PJ}^{-1}, and r_{PJ3}^{-1}. The transfer rule corresponding to transformation rule PJ3 is simple: Simply remove the parallel parts. Transfer rules r_{PP}^{-1} and r_{PJ}^{-1} move tokens from the extended part to the superclass part. Note that as long as the designer sticks to the inheritance preserving transformation rules, the transfer rules can be generated automatically, so no complicated migration schemes have to be designed.

5.3 Related work on interorganizational workflow

Most of today's commercial workflow systems use a centralized enactment service. Therefore, many of the research prototypes such as MENTOR (University of Saarland at Saarbrucken), METEOR (University of Georgia), MOBILE

(University of Erlangen), Panta Rhei (University of Klagenfurt), and WASA (University of Muenster) focus on distribution aspects. The MENTOR system is based on state charts partitioned into fragments, which are distributed under the supervision of Tuxedo, a TP monitor. The METEOR system is entirely based on CORBA to provide a platform independent environment. It also supports interoperability mechanisms like SWAP and JFLOW. Moreover, the METEOR3 model introduces the notion of foreign task vs. native tasks. A foreign task refers to a task whose realization is unknown to the workflow designer, whereas the implementation details are known for a native task. Another important feature for E-commerce are channels (also called sink nodes) that are used to specify communication or synchronization between two independent workflows. An interesting project focussing on workflow technology in E-commerce is the WIDE Project (`http://dis.sema.es/projects/WIDE/`). Its goal is to enable interorganizational workflows across multiple platforms by linking geographically separated applications including IBM's MQSeries Workflow, SAP R/3, Opera, and Structware. Other projects focussing on interorganizational workflows are Cross-Flow (`http://www.crossflow.org/`), MariFlow, and ACEFlow.

6 Conclusion

Modeling of enactment of interorganizational workflows is a central issue. In this paper we used reference nets as the basic modeling technique. They allow for distributed execution and provide the conceptual and practical means to model the different perspectives of an enactment service in a natural way. Renew enables workflow designers to build appropriate prototypes and directly model the different perspectives. Users can now identify their different perspectives directly within models. Hence the interpretation becomes relatively easy. Furthermore the concept of mobility is nicely expressed by moving the appropriate object-nets which are directly related to some conceptual objects of users. Here we only discussed the movement of workflows with their related data and the movement of resources. However, migration can be implemented for every perspective. The conceptual basis is the same. Problems that could occur due to different kinds of implementations are covered by the inheritance concept which can be used to check that casting between the different types is done in the correct way.

References

[Aal98] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[Aal99] W.M.P. van der Aalst. Interorganizational Workflows: An Approach based on Message Sequence Charts and Petri Nets. *Systems Analysis - Modelling - Simulation*, 34(3):335–367, 1999.

[AB97] W.M.P. van der Aalst and T. Basten. Life-cycle Inheritance: A Petri-net-based Approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, Berlin, 1997.

[AB99] W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An approach to tackling problems related to change. Computing Science Reports 99/06, Eindhoven University of Technology, Eindhoven, 1999. in print.

[BA99] T. Basten and W.M.P. van der Aalst. Inheritance of Dynamic Behaviour: Development of a Groupware Editor. In G. Agha, F. De Cindo, and G. Rozenberg, editors, *Advances in Petri Nets: Concurrent Object-Oriented Programming and Petri Nets*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1999. in print.

[Bas98] T. Basten. *In Terms of Nets: Systems Design with Petri Nets and Process Algebra.* PhD thesis, Eindhoven University of Technology, Eindhoven, 1998.

[Bus99] C. Bussler. Workflow class inheritance and dynamic workflow class binding. In W. van der Aalst, J. Desel, and R. Kaschek, editors, *Proceedings of the Workshop Software Architectures for Business Process Management at the 11th Conference on Advanced Information Systems Engineering CAiSE*99*, Heidelberg, Germany, June 1999. Report No. 390, University of Karlsruhe.

[CH92] Søren Christensen and Niels Damgaard Hansen. Coloured Petri Nets Extended with Channels for Synchronous communication. Technical Report DAIMI PB–390, Computer Science Department, Aarhus University, DK-8000 Aarhus C, Denmark, April 1992.

[EN93] C.A. Ellis and G.J. Nutt. Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, Berlin, 1993.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[Hol95] Tom Holvoet. Agents and petri nets. In O. Herzog, W. Reisig, and R. Valk, editors, *Petri Net Newsletters*, number 49 in Petri Net Newsletters, 1995.

[JB96] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation.* International Thomson Computer Press, 1996.

[Kou95] T.M. Koulopoulos. *The Workflow Imperative.* Van Nostrand Reinhold, New York, 1995.

[KW99] Olaf Kummer and Frank Wienberg. Renew homepage. URL: `http://www.renew.de`, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 1999.

[Lak95] C.A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *16th International Conference on the Application and Theory of Petri Nets*, number 935 in Lecture Notes in Computer Science, pages 278–297, Torino, Italy, 1995. Springer.

[Law97] P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition.* John Wiley and Sons, New York, 1997.

[VA] E. Verbeek and W.M.P. van der Aalst. Woflan Home Page. http://www.win.tue.nl/ woflan.

[Val87] Rüdiger Valk. Modeling of task-flow in systems of functional units. Technical Report FBI-HH-B-124/87, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 1987.

[Val98] Rüdiger Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In Jörg Desel, editor, *19th International Conference on Application and Theory of Petri nets*, number 1420 in LNCS, Berlin, 1998. Springer-Verlag.