International Journal of Cooperative Information Systems
© World Scientific Publishing Company

PROCLETS: A FRAMEWORK FOR LIGHTWEIGHT INTERACTING WORKFLOW PROCESSES

W.M.P. VAN DER AALST*
Department of Technology Management, Eindhoven University of Technology, P.O. Box 513
Eindhoven, NL-5600 MB, The Netherlands

P. BARTHELMESS
Department of Computer Science, University of Colorado at Boulder, Campus Box 430
Boulder, CO 80309-0430, USA

C.A. ELLIS
Department of Computer Science, University of Colorado at Boulder, Campus Box 430
Boulder, CO 80309-0430, USA

J. WAINER
Department of Computer Science, State University of Campinas, Caixa Postal 6176
Campinas, 13083-970, Brazil

Received (to be inserted
Revised by Publisher)

The focus of traditional workflow management systems is on control flow *within one* process definition. The process definition describes how a single case (i.e., workflow instance) in isolation is handled. For many applications this paradigm is inadequate. Interaction between cases to support communication and collaboration is at least as important. This paper introduces and advocates the use of interacting *proclets*, i.e., lightweight workflow processes. By promoting interactions to first-class citizens it is possible to model complex workflows in a more natural manner. In addition, the expressive power and flexibility are improved compared to the more traditional workflow modeling languages.

Keywords: Workflow management systems; Proclets; process modeling; Petri nets; interaction mechanisms; workflow agents

1. Introduction

In the last decade many workflow management systems have become available.[1] These systems allow for the explicit representation and support of business processes and avoid the need to re-code applications every time a business process changes. As the workflow paradigm continues to infiltrate organizations that need to cope with complex administrative processes, it is becoming apparent that the available workflow management systems have difficulties dealing with the increas-

*Part of this work was done while visiting the Department of Computer Science of the University of Colorado.

ingly dynamic and inter-organizational nature of today's business processes.[2] As we will argue in this paper, one of the core problems of the current generation of workflow languages and tools is the focus on isolated case-based processes.

Perspectives that are relevant for workflow modeling and workflow execution are: (1) *control-flow* (or process) perspective, (2) *resource* (or organization) perspective, (3) *data* (or information) perspective, (4) *task* (or function) perspective, and (5) *operation* (or application) perspective. (These perspectives are similar to the perspectives given by Jablonski.[3]) In this paper we primarily focus on the control-flow perspective. This does not imply that the other perspectives are less relevant. However, the problems addressed in this paper are mainly related to the control-flow perspective. In traditional workflow management systems, the control-flow perspective of a workflow is described by one *workflow process definition* (also called workflow schema). A workflow process definition specifies which tasks need to be executed and in what order (i.e., the routing or control flow). A *task* is an atomic piece of work. Workflow process definitions are instantiated for specific *cases* (i.e., workflow instances). Examples of cases are a request for a mortgage loan, an insurance claim, a tax declaration, an order, or a request for information. Since a case is an instantiation of a process definition, it corresponds to the execution of concrete work according to the specified routing.

One of the authors of this paper has been involved in a detailed investigation into the expressive power of existing workflow products.[4] This research is based on a set of *workflow patterns*. Each pattern corresponds to a typical control-flow construct frequently encountered in real-life workflow processes. These patterns have been used to evaluate 14 workflow products (COSA, Visual Workflow, Forté Conductor, Meteor, Mobile, MQSeries/Workflow, Staffware, Verve Workflow, I-Flow, InConcert, Changengine, SAP R/3 Workflow, Eastman, and FLOWer) and the results can be found on the workflow patterns web site.[5] This research shows that most of the workflow management systems support less than half of the workflow patterns. Clearly, workflow languages limited to traditional building blocks such as the AND/XOR-split/join are inadequate for supporting real-life processes.

Today's workflow management systems predominantly focus on the control-flow *within one* process definition. This assumes that a workflow process can be modeled by specifying the *life-cycle of a single case in isolation*. For many real-life applications this assumption is too restrictive. As a result, the workflow process is changed to accommodate the workflow management system, the control-flow of several cases is artificially squeezed into one process definition, or the coordination amongst cases is hidden inside custom built applications. Consider for example an engineering process of a product consisting of multiple components. Some of the tasks in this engineering process are executed for the whole product, e.g., the task to specify product requirements. Other tasks are executed at the level of components, e.g., determine the power consumption of a component. Since a product can have a variable number of components and the components are engineered concurrently, it is typically not possible to squeeze this workflow into one process definition. This is a direct consequence of the fact that, in most workflow management systems, the degree of parallelism is fixed in a workflow process definition, i.e., it is not possible to concurrently instantiate selected parts of the workflow process a variable number of times. Using iteration one can instantiate parts a variable number of times. However, this results in the sequential execution of inherently parallel tasks.

To solve these problems, we propose an approach based on *proclets*, *performatives* and *channels*. Proclets are lightweight processes. Typically, a proclet represents only one aspect or one element of the whole workflow. The interaction between proclets is modeled explicitly, i.e., proclets can exchange structured messages, called performatives, through channels. By adopting this approach the problems related to purely case-based processes can be avoided.

The remainder of this paper is organized as follows. First, we motivate our approach by clearly identifying the problems encountered when modeling the paper selection process of a conference. Then we present the framework which is based on Petri nets[6,7] and inspired by concepts originating from object-orientation[8,9], conceptual modeling[10], agent-orientation[11], and the language/action perspective[12,13,14,15]. In Section 4, we model the reviewing process of a conference using our framework. Section 5 presents an additional example, the hiring of new employees. Finally, we compare the framework with existing approaches and conclude with our plans for future research.

2. Motivating Example: Paper Selection Process of a Conference

The process of selecting papers for a conference presents features that challenge existing workflow modeling languages. In brief, the goal of this process is to select some papers out of a normally larger set, based on different criteria (e.g., quality, minimum and maximum number of papers). A set of people is invited to act as program committee members, these people can accept or decline, additional people can be invited, and, finally, a call for papers is issued to prospective authors. These authors submit papers that are then subject to review by peers (invited by program committee members) and finally a selection is made. A very brief and abstract sequence of steps would be:

- Invite program committee (PC) members: these are going to be responsible for the management of reviews.
- Issue a call for papers: this step announces the upcoming conference and asks for submissions.
- Receive the submissions and check them: papers are accepted up until a deadline. Submissions are checked for consistency with conference standards, and so on.
- Distribution: after the submission deadline, each paper is assigned to multiple PC members. These PC members will be responsible for finding reviewers for

the papers assigned to them. The goal is to obtain at least a minimum number of reviews, by different people, for each paper.

- Review: reviewing starts with the assignment of a reviewer by a PC member. The paper is made available for the reviewer and after a while a review is produced.
- Selection: after the reviews are completed, papers are compared and ranked according to the reviewer's recommendations and other subjective criteria (e.g., desired number of papers, acceptable quality threshold).
- Notification: authors are notified either of acceptance or rejection of their papers. In case of acceptance, final versions have to be sent in by the authors.
- Publication: final versions are assembled and sent for publication.

The process is complicated by a series of factors, which we list in a non-exhaustive way:

- Prospective PC members and reviewers may accept or reject the invitation to join the committee and to review one or more papers, respectively. Replacements for those that rejected the invitation need to be found.
- Reviewers can fail to return the reviews on time. They may either declare that they are not going to meet the deadline or simply forget about the deadline altogether. As a result, some of the papers may lack enough reviews to allow their fair evaluation.
- The distribution of the papers takes into account varied criteria, such as the number of submitted papers, the number of available PC members, preferences and areas of expertise of PC member's groups, balance of the work load assigned to each PC member (as compared to their availability) and so on. The decision of how to split the papers needs to take into account the whole set of available papers and cannot be performed in isolation.
- Selection is a yet more subjective task. Once again, this task can only be performed on the whole set of available papers. Paper quality needs to gauged against the quality of all the remaining papers, to a certain extent, or at least to a set of related papers. If two or more papers discuss the same topic from different or opposing perspectives, this needs to be taken into account. Other factors, such as the minimum and maximum number of desired papers (i.e., empty slots) also influence this task.

A modeler faces many problems translating these requirements. A first basic question is what is to be considered the case[†] - the submission, the review, the "empty

[†]Workflow instance.

slot" in the conference that one wants to fill with a good quality paper, or is the case the whole set of slots? The choice of each of these as the unit of modeling causes problems. A closer examination of the tasks of such a process reveals that while some of the tasks operate on each submitted paper individually, others are based on the whole set of papers, and others still are related to individual reviews. While tasks such as *receiving* and *checking*, for example, can be conducted at the level of individual papers, tasks such as *distribution to program committee members* and *final selection* are based on the whole set of papers. *Review* tasks operate on each of the multiple reviews that are produced for each paper.

The class diagram (Figure 1) shows that different tasks rely on information that is at different levels of aggregation - some of the tasks operate at the conference level, that groups all papers, others at the paper level, and others yet at the lower level of a single review. The choice of any of the possible aggregations as the main one introduces problems whenever we have to deal with the others. One of the major obstacles is, therefore, how to reconcile these multiple perspectives into one model.

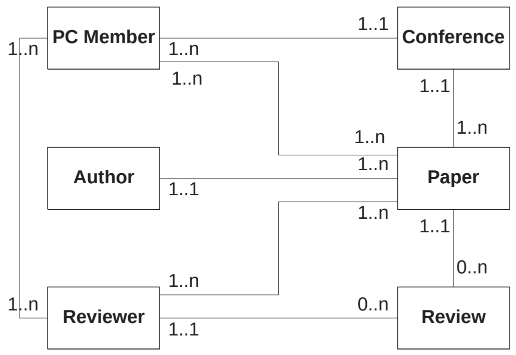


Figure 1: Review process class diagram.

Lacking the power to express differences in aggregation, most workflow management systems force one to depict the process at an arbitrarily chosen level (usually the paper level), essentially ignoring the issues that are relevant at the conference and to some extent at the review levels. The resulting models present some important shortcomings:

- The models are artificially flattened, being unable to account for the mix of different perspectives that coexist in the real process. Given that workflow enactment is guided by what is specified by the process model, the missing perspectives will have to be handled and coordinated manually by the users themselves, without further help from the system.
- Batch-oriented tasks are typically not supported. Batch-oriented tasks are those that are based on groupings of lower aggregation elements, e.g., the

whole set of papers during distribution and selection, or the set of reviews for a paper, while deciding if enough reviews are available. In other words, it is usually not possible to handle higher aggregation tasks using lower aggregation instances, e.g., conference level tasks within a paper level case.

- Handling lower aggregation tasks within higher aggregation ones is also hard in most languages. Launching and then synchronizing a variable number of reviews (lower aggregation) from a paper centered case, for instance, cannot be usually represented in most languages.
- The interactions with the environment are usually abstracted away as well. An important aspect in many processes is the exchange of messages between the entities. Reviewers, for instance: receive invitations to review papers; respond to them by either accepting or rejecting; must be notified of approaching deadlines; send their completed reviews or sometimes send notification of inability to complete reviews. These interactions need to be reflected in the process model, but usually are not.

Conference review is not an atypical example, in the sense that one encounters similar problems very frequently in other areas as well. We next list just a few of the innumerable real world examples where interactions between instances and different levels of aggregation play a strong role:

- In engineering processes: processing of subparts may impact one or more higher level components that make use of this common subpart. Conversely, decisions at the higher level component processes may have an impact on subpart processes. For example, an approaching deadline for a higher level component may cause interruption of the process of certain subparts.
- In software development: software modules are composed of submodules, which in turn may be composed of sub-submodules and so on. Considerations at higher or lower levels of aggregation may influence other levels, e.g., the discovery of some specification flaw at a lower level may have a ripple effect on a variable number of modules at all other levels. Code that is shared by multiple versions also introduces interactions that are hard to model.
- Processing of insurance claims: some claims may refer to the same accident. Even though they may start out as independent instances, at some point in time it is desirable that all related claims be merged so that a uniform decision can be reached.
- Hiring new people: some job applications are received in response to an advertised open position. Candidates have to be evaluated and ranked with respect to each other. Again, the interactions between the applications are most relevant. Some applications are sent in independently of open positions.

In this situation, interesting applications may cause a position to be specially created. Again, there is a strong interaction between two perspectives, that of the application and of the position. Sometimes one is the central one, sometimes it is the other.

In summary, we see as limitations of current modeling formalisms 1) the fact that one is usually forced to choose to represent a process at one single level, even when a problem space consists of entities with varying aggregations, 2) that the interactions with the environment cannot be made explicit, even though a subjacent model may be (and usually is) assumed.

3. Framework

The examples given in the previous section show that today's workflow management systems typically have problems dealing with workflow processes that are not entirely *case-oriented*. Squeezing the control flow of a workflow process into a single process definition often results in unreadable workflow specifications where essential parts of the control flow are hidden inside custom made application software. In fact, there are plenty of examples where the workflow process is changed in order to fit the workflow management system. Clearly, this is undesirable: Workflow technology should support rather than dictate work processes.

Inspired by these problems, we have developed a new framework for modeling workflows. This framework is based on *proclets*. A proclet can be seen as a lightweight workflow process equipped with a knowledge base containing information on previous interactions. One can think of proclets as objects equipped with an explicit life-cycle (in the object-oriented sense[8,9]) or active documents (i.e., documents aware of tasks and processes[16]). Proclets interact via *channels*. A channel is the medium to transport messages from one proclet to another. The channel can be used to send a message to a specific proclet or a group of proclets (i.e., multicast). Based on the properties of the channel different kinds of interaction are supported, e.g., push/pull, synchronous/asynchronous, and verbal/non-verbal. In order for proclets to find each other, there is a *naming service*. The naming service keeps track of registered proclets and can be queried by any proclet. The concepts *proclet*, *channel* and *naming service* constitute a framework for modeling workflow processes (see Figure 2).

Compared to existing workflow modeling languages, complex case-based workflow definitions describing the control flow of an entire process are broken up into smaller interacting proclets, i.e., there is a shift from control to communication. The framework is based on a solid process modeling technique (Petri nets[6,7]) extended with concepts originating from object-orientation[8,9], agent-orientation[11], and the language/action perspective[12,13,14,15].

In the remainder of this section we present the four main components of our framework: *proclets*, *knowledge base*, *channels*, *naming service*, and *actors*.

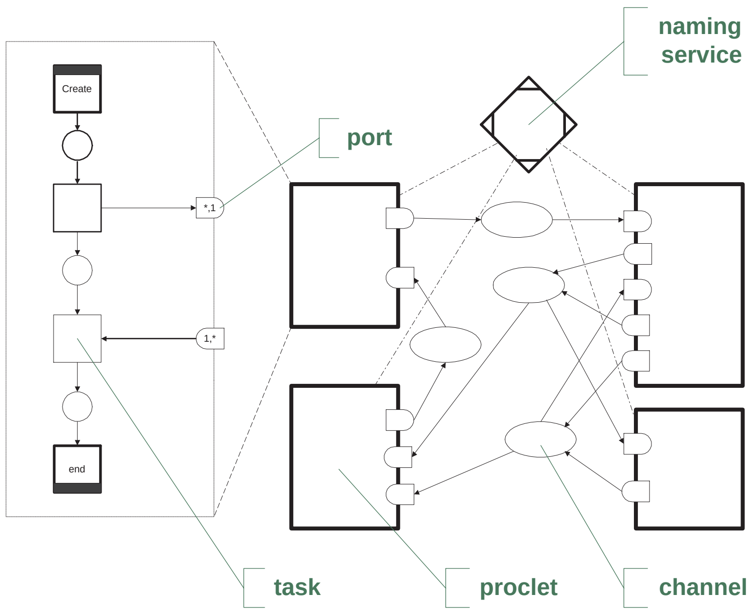


Figure 2: Graphical representation of the framework.

3.1. Proclets

A *proclet class* describes the life-cycle of *proclet instances*. A proclet class can be compared to an ordinary workflow process definition or workflow type.[3] The class describes the order in which tasks can or need to be executed for individual instances of the class, i.e., it is the specification of a generic process. Proclet instances can be created and destroyed, and are executed according to a class specification. At any moment a proclet instance has a state. When no confusion is possible we will simply use the term “proclet” instead of “proclet class” and/or “proclet instance”.

To define proclets, we introduce some preliminaries including some basic Petri net concepts and terminology.

To specify proclet classes, we use a graphical language based on *Petri nets*. Petri nets are an established tool for modeling and analyzing workflow processes.[17,18,19,20] On the one hand, Petri nets can be used as a design language for the specification of complex workflows. On the other hand, Petri net theory provides for powerful analysis techniques which can be used to verify the correctness of workflow procedures.[6,7] A (classical) Petri net is a directed bipartite graph with two node types called *places* and *transitions*. The nodes are connected via directed *arcs*. Connections between two nodes of the same type are not allowed. Places are represented by circles and transitions by rectangles. A place p is called an *input place* of a transition t iff there exists a directed arc from p to t. Place p is called an *output place* of transition t iff there exists a directed arc from t to p. At any time a place contains zero of more *tokens*, drawn as black dots. The *state*, often referred to as marking, is the distribution of tokens over places. The number of tokens may change during the execution of the net. Transitions are the active components in a Petri net: they change the state of the net according to the following *firing rule*:

(1) A transition t is said to be *enabled* iff each input place p of t contains at least one token.

(2) An enabled transition may *fire*. If transition t fires, then t *consumes* one token from each input place p of t and *produces* one token in each output place p of t.

Petri nets can move from one state to another by firing enabled transitions. A state s is *reachable* if there is a sequence of transition firings which leads from the current state to state s. A Petri net in a given state is *safe* if for any reachable state no place contains multiple tokens, i.e., the number of tokens per place is limited to 1. A Petri net in a given state is *live* if for any reachable state s and for any transition t it is possible to reach a state from s such that t is enabled. A transition t is called *dead* if there is no reachable state enabling t. Reachable, safe, live, and dead are standard concepts which can be found in any textbook on Petri nets.[6]

In this paper, we use a specific subclass of Petri nets. This subclass corresponds to the so-called class of sound WF-nets.[17‡] A WF-net has source and sink transitions: A source transition has no input places and a sink transition has no output places. Every node (i.e., place or transition) is on a path from some source transition to some sink transition. Moreover, any WF-net is connected, i.e., the network structure cannot be partitioned in two unconnected parts. A WF-net becomes activated if one of the source transitions fires. In the remainder we assume that a WF-net becomes activated only once (single activation). A WF-net is called *sound* if and only if the following requirements are satisfied:

(1) *safeness*: each state reachable under the single activation assumption is safe.

(2) *proper completion*: firing one of the sink transitions empties the net, i.e., after firing a sink transition no tokens are left.

(3) *completion option*: from any reachable state it is possible to reach a state which enables one of the sink transitions, i.e., termination is always possible.

(4) *dead transitions*: there are no dead transitions.

These four requirements are quite reasonable in the context of workflow management: It should always be possible to terminate properly, there should be no parts which cannot be activated, and, since the WF-net will model one proclet instance, it should be safe. Soundness can be verified using state-of-the-art analysis techniques.[17] Based on these techniques we have developed a workflow verifier called Woflan.[21] See the Woflan website for more information.[22]

[‡]For the readers familiar with WF-nets: For notational convenience we omit the unique source and sink place.[17]

Most workflow modeling languages primarily focus on control flow inside one process definition and (partly) abstract from the interactions between process definitions, i.e., coordination is limited to the scope of the process definition and communication and collaboration are treated as second-class citizens. Therefore, our framework explicitly models interactions between proclets. The explicit representation of interactions is inspired by the *language/action perspective*[15,14] which was introduced in the field of information systems by Flores and Ludlow[12] in the early 1980's and is rooted in *speech act* theory.[23] In contrast to traditional views of "data flow" the language/action perspective emphasizes what people do while communicating; how they create a common reality by means of language and how communication brings about a coordination of their activities. The need for treating interaction as first-class citizens is also recognized in the agent community[11]. Emerging agent communication languages such as KQML[24] demonstrate this need.

Inspired by these different perspectives on interaction, we use *performatives* to specify communication and collaboration among proclets. A performative is a message exchanged between one sender proclet and one or more receiver proclets. A performative has the following attributes:

(1) *time*: the moment the performative was created/received.

(2) *channel*: the medium used to exchange the performative.

(3) *sender*: the identifier of the proclet creating the performative.

(4) *set of receivers*: the identifiers of the proclets receiving the performative, i.e., a list of recipients.

(5) *action*: the type of the performative.

(6) *content*: the actual information that is being exchanged.

The role of these attributes will be explained later. At this point, it is important to note the action attribute. This attribute can be used to specify the illocutionary point of the performative. The five illocutionary points identified by Searle[23] (assertive, directive, commissive, declarative, expressive) can be used to specify the intent of the performative. Examples of typed performatives identified by Winograd and Flores are request, offer, acknowledge, promise, decline, counter-offer or commit-to-commit.[15] In this paper, we do not restrict our model to any single classification of performatives (i.e., a fixed set of types). However, at the same time we stress the importance of using the experience and results reported by researchers working on the language/action perspective.

Proclets combine performatives and sound WF-nets. A *proclet class PC* is defined as follows:

(1) PC has a *unique name*. This name serves as a unique identification of the class which we will refer to as *class_id*.

(2) *PC* has a *process definition* defined in terms of a *sound WF-net*. The transitions correspond to *tasks* and the places correspond to *state conditions*.

(3) *PC* has *ports*. Ports are used to interact with other proclets. Every port is connected to one transition.

(4) Transitions can send and receive *performatives* via ports. Each port has two attributes: (a) its *cardinality* and (b) its *multiplicity*. The cardinality specifies the number of recipients of performatives exchanged via the port. The multiplicity specifies the number of performatives exchanged via the port during the lifetime of any instance of the class.

(5) *PC* has a *knowledge base* for storing these performatives: Every performative sent or received is stored in the knowledge base.

(6) Tasks can query the knowledge base. A task may have a *precondition* based on the knowledge base. A task is enabled if (a) the corresponding transition in the WF-net is enabled, (b) the precondition evaluates to true, and (c) each input port contains a performative.

(7) Tasks connected to ports have *post conditions*. The post condition specifies the outcome of the task in terms of performatives generated for its output ports. The performatives which are generated may depend upon information obtained from the *naming service* (i.e., proclet identifiers).

A proclet class is a generic definition, i.e., it does not describe the behavior and properties of a specific proclet. Proclet (instances) are created by instantiating the proclet class and have a *unique identification* which we will refer to as *proc_id*. Note that for a concrete proclet all elements, i.e., the process definition, knowledge base, ports, and tasks, are instantiated. For example, the tokens in the WF-net specifying the process definition refer to one proclet instance, i.e., tokens of different proclet instances are *not* merged into one WF-net. (Recall that a sound WF-net is safe.) Moreover, each proclet instance has its own private knowledge base. However, proclet instances can *share* performatives with all other instances of the same class. This means that part of the knowledge base is *public* and part of the knowledge base is *private*. The public part is identical for all instances of the class, i.e., effectively this part resides at the class level. The private part exclusively resides at the instance level. Whenever a performative is sent or received, the corresponding proclet decides whether it should be stored in the public or in the private part.

A performative has by definition one sender, but can have multiple recipients. The sender is always represented by a proc_id, i.e., the identifier of a proclet instance. However, the list of recipients can be a mixture of proc_id's and class_id's, i.e., one can send performatives to both proclet instances and proclet classes. A performative sent to a proclet class is received by all proclet instances of that class. Note that the naming service can be used to obtain the desired proclet identifiers (cf. Section 3.4).

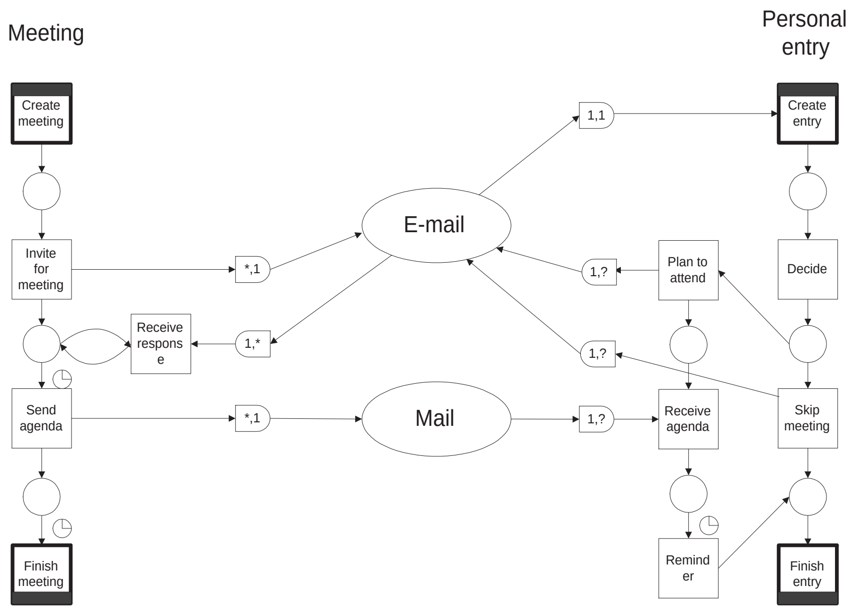


Figure 3: Example of two proclet classes: *Meeting* and *Personal entry*.

To illustrate the framework we use the example shown in Figure 3. There are two proclet classes. Both classes are used to organize meetings. Proclet class *Meeting* is instantiated once per meeting. Proclet class *Personal entry* is instantiated for every potential participant of a specific meeting. The instance of class *Meeting* first sends an invitation to all potential participants. The proc_id's are used to multicast the invitation performative to a specified set of instances of class *Personal entry*. Note that the cardinality of the port connected to task *Invite for meeting* is denoted by a star $*$. This star indicates that the invitation is sent to an arbitrary number of potential participants, i.e., the performative has multiple recipients. We will use $*$ to denote an arbitrary number of recipients, $+$ to denote at least one recipient, 1 to denote precisely one recipient, and *?* to denote no or just a single recipient. Performatives with no recipients are considered not to have occurred, i.e., only performatives with a positive number of recipients are registered in the knowledge base. The multiplicity of the output port connected to task *Invite for meeting* is denoted by the number 1. This means that during the lifetime of an instance of class *Meeting* exactly one performative is sent via this port. The invitation performative is sent though the channel *E-mail* (The role of channels is explained in Section 3.3). The performative creates a proclet for each recipient, i.e., creation task *Create entry* is triggered. Creation tasks are depicted by squares with a black top. The input port connected to *Create entry* has cardinality 1 and multiplicity 1. Every input port has by definition cardinality 1, i.e., from the perspective of the receiving proclet there is only one proclet receiving the performative. Input ports connected to a creation task (i.e., a source transition) have by definition a multiplicity of 1 or ?: An instance can be created only once. Since there is just one creation task in *Personal entry*, the multiplicity is 1. After an instance of the class *Personal entry* is created a decision is made (task *Decide*). Based on this decision either task *Skip meeting* or *Plan to attend* is executed. In both cases a performative is sent to the instance of the proclet class *Meeting*. The performative is either a confirmation (*Plan to attend*) or a notification of absence (*Skip meeting*). Note that each instance of the class *Personal entry* sends such a performative. These performatives are sent through channel *E-mail*. Note that the ports connected to *Plan to attend* and *Skip meeting* both have cardinality 1 (i.e., one recipient) and multiplicity ? (one performative is sent via one of the two ports). Task *Receive response* is executed once for every "confirmation/notification of absence" performative. Therefore, the corresponding port has multiplicity $*$. After some time, as indicated by the clock symbol[17], task *Send agenda* is executed. In this small example we assume that all potential participants respond before this time-out occurs. *Send agenda* generates one performative: the agenda of the meeting. This performative is sent to all proclets that confirmed the invitation. This performative has multiple recipients, i.e., the cardinality of the corresponding output port is $*$. Since the agenda is sent only once the multiplicity is 1. The proclets that confirmed the invitation receive the agenda (task *Receive agenda*) and a timer for the task *Reminder* is set. Finally, all proclets are destroyed by executing the finishing tasks *Finish meeting* and *Finish entry*. The finishing tasks (i.e., sink transitions) are depicted by squares with a black bottom.

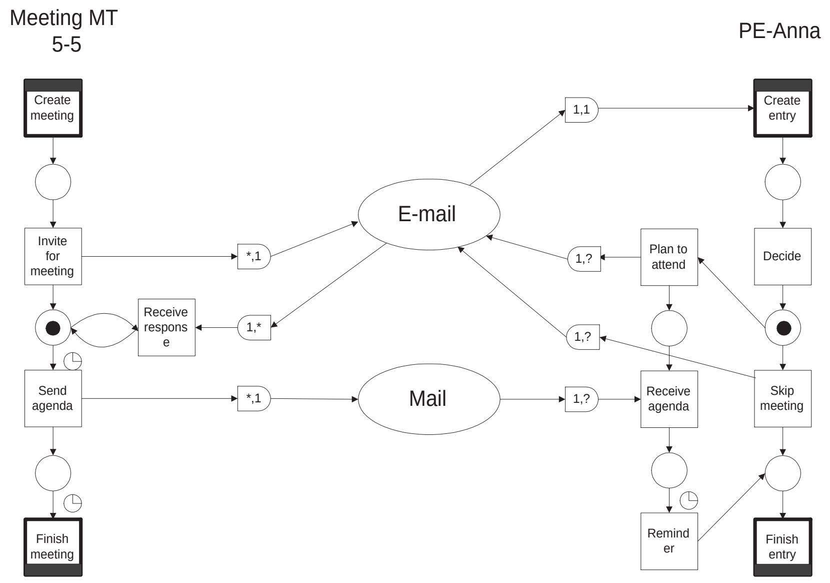
3.2. Knowledge base

A proclet can use knowledge in its knowledge base to make routing decisions. This knowledge can range from simple data to beliefs about other proclets. Building a good knowledge base is not a trivial task. First of all, there has to be an ontology to characterize the intended meaning of terms and concepts. Then, the scope and knowledge acquisition process have to be identified.

In this paper, we use a more restrictive definition of a knowledge base. We simply see the knowledge base as a set of relevant performatives. The knowledge base of a proclet contains all the performatives that it sent and received. Some of these performatives are visible to all proclets in the class. These performatives are called *public*. The remaining performatives are *private* and only visible by the corresponding proclet.

The knowledge base could be extended with more knowledge than performatives. However, to simplify the presentation of the proclet framework, we use this restricted view.

To illustrate the role of the knowledge base, we use the model presented in Figure 3. If we instantiate this model, we get concrete proclets. Figure 4 shows two proclets: A proclet of class *Meeting* and a proclet of class *Personal entry*. The



time	channel	sender	receivers	action	content	scope	direction
1000	-	-	Meeting MT 5-5	Create	…	Private	IN
1010	E-mail	Meeting MT 5-5	PE-John PE-Suzan PE-Clark PE-Anna	Request	Will you come to the meeting on May 5th?	Private	OUT
1020	E-mail	PE-John	Meetin g MT 5-5	Promise	I will come.	Private	IN
1030	E-mail	PE-Suzan	Meeting MT 5-5	Decline	Sorry, I will not be there	Private	IN

time	channel	sender	receivers	action	content	scope	direction
1010	E-mail	Meeting MT 5-5	PE-Anna	Request	Will you come to the meeting on May 5th?	Private	IN

Figure 4: A proclet of class *Meeting* and a proclet of class *Personal entry*.

proclet *Meeting MT 5-5*, i.e., the meeting of the management team planned on the fifth of May, is in the state after sending the invitation and before sending the agenda. The proclet *PE-Anna* corresponds to the personal entry of Anna who is a member of the management team. Anna is about to decide whether she will attend the meeting of the management team on the fifth of May.

time	channel	sender	receivers	action	content	scope	direction
1000	-	-	Meeting MT 5-5	Create	…	Private	IN
1010	E-mail	Meeting MT 5-5	PE-John PE-Suzan PE-Clark PE-Anna	Request	Will you come to the meeting on May 5th?	Private	OUT
1020	E-mail	PE-John	Meeting MT 5-5	Promise	I will come.	Private	IN
1030	E-mail	PE-Suzan	Meeting MT 5-5	Decline	Sorry, I will not be there	Private	IN
1035	E-mail	PE-Anna	Meeting MT 5-5	Promise	OK, I will be there	Private	IN
1040	E-mail	PE-Clark	Meeting MT 5-5	Promise	I will definitely come.	Private	IN
1100	Mail	Meeting MT 5-5	PE-John PE-Clark PE-Anna	Inform	9.00 Opening 9.30 Presentation 10.00 Discussion	Private	OUT

time	channel	sender	receivers	action	content	scope	direction
1010	E-mail	Meeting MT 5-5	PE-Anna	Request	Will you come to the meeting on May 5th?	Private	IN
1035	E-mail	PE-Anna	Meeting MT 5-5	Promise	OK, I will be there	Private	OUT
1100	Mail	Meeting MT 5-5	PE-Anna	Inform	9.00 Opening 9.30 Presentation 10.00 Discussion	Private	IN

Figure 5: The knowledge base of both proclets after sending the agenda.

Figure 4 shows the knowledge base of both proclets. The knowledge base of proclet *PE-Anna* contains only one performative which corresponds to the invitation to the meeting. This performative was received at time 1010 via E-mail and is only visible by *PE-Anna*. This performative triggered the creation of the personal entry for Anna. The knowledge base of proclet *Meeting MT 5-5* contains four performatives. The first performative is the creation of *Meeting MT 5-5*. The second one contains the invitation to the four members of the management team, i.e., John, Suzan, Clark, and Anna. The action associated to this performative is a request. The other two performatives correspond to responses to this request. John promises to come and Suzan declines the invitation. If Anna declines or accepts the invitation to come to the meeting, a performative is added to the knowledge base of *Meeting MT 5-5*. Assume that Suzan is the only one not attending the meeting. The resulting situation after sending the agenda is shown in Figure 5.

Figures 4 and 5 illustrate the evolution of two knowledge bases. The example does not show the actual use of this knowledge. As indicated before, pre and post conditions can be based on the knowledge stored in the knowledge base. The simple choice between two alternative tasks may be based on the presence of a given performative in the knowledge base. For example, we could extend the proclet class *Meeting* with an additional task named *cancel_meeting*. This task would have the precondition that there should be less than two performatives of type *Decline*, i.e., a majority of the management team has to be present. In this paper we do not propose a concrete syntax for these pre and post conditions: Different types of languages can be used for this purpose.

3.3. Communication Channels

Communication channels are used to link proclets. Channels transmit messages containing performatives from sending proclets to receiving proclets. There are many different categories of channels defined by channel properties such as medium type, reliability, security, synchronicity, closure, and formality. These properties are briefly explained:

- *Medium Type*
  This can be point-to-point or broadcast, or some form of limited multicast. Recall that performatives can be sent to an individual proclet instance (point-to-point), a set of proclets (multicast), or an entire proclet class (broadcast). Common media include postal mail, telephone, and electronic mail. Different media satisfy different communication requirements. We are also concerned with media of face-to-face communication such as sound waves of spoken voice, gestures, and body language. The framework presented in this paper, assumes that there is only one sending proclet. However, there are situations where a group effort results in a single performative (e.g., orchestral performances). In fact there are many examples that could not be accomplished by a single person or proclet (e.g., collaborations modeled as single acts such as lifting a heavy object). Such group efforts can be modeled by introducing a so-called proxy proclet. This proclet coordinates and consolidates the group effort.

- *Reliability*
  Some channels are very reliable; some are unreliable. For some electronic channels, we assume that the technology is robust, and that error detection and retransmission are implemented at lower layers of the communication protocols. In this case, we need not be concerned with these details in our higher level modeling. Thus, channels built upon TCP/IP are more reliable than those built upon UDP. A problem of dial-in data channels in some lesser developed countries is that the channel (the phone lines) are inherently unreliable. Thus, sometimes the data gets sent, and sometimes not. Similar unreliability is sometimes exhibited by postal services. A different channel available from the postal service is registered mail, where the cost of mailing a letter is higher, and the reliability is also higher.

- *Security*
  At times the content of a performative is considered to be quite valuable and secret. In such cases, the transmission should be via highly secure channels. In electronic transmission, encoding and encryption are sometimes used to implement secure channels.

- *Synchronicity*
  This is concerned with the time delay of message delivery and acknowledgment. Some channels are used for real time communications in which each party expects to get rather immediate feedback from recipient parties. This requires synchronous channels. Face-to-face spoken conversation falls into this category. In other cases, the expectation is that the recipient will not instantaneously receive the message content. In the case of an asynchronous channel, the sender usually is not waiting for an immediate response. For example, when email is sent, there is usually no expectation of immediate response. When a UNIX talk session is initiated, there is expectation of immediate response.

- *Closure*
  Channels can be classified as open or closed channels. When a channel is open, the sender does not know exactly who, and how many recipients are connected. When a channel is closed, the exact identity of all recipients is specified in advance. A radio broadcast, and a notice posted on a bulletin board are respectively examples of synchronous and asynchronous communications in which the medium is open because the senders do not exactly know who are their recipients.

- *Formality*
  Some channels convey much more formality in the messages delivered than others. Performatives can be very formally specified, or can be informal and flexible. Generally, business letters are much more formal than chat rooms. A scheduled meeting with a rigid agenda is much more formal than a casual conversation over coffee. A careful record is kept of formal channel transmissions, whereas informal channels are usually not recorded; they are "off the record."

The various properties of the communication channels are often neglected in existing modeling languages. Consider for example asynchronous communication versus synchronous communication. Setting the date for a meeting through synchronous communication has the drawback that it may not be possible to reach the participants at a given time. However, if it is possible to reach a participant, then it is possible to set a date immediately. The latter is not possible through asynchronous communication. We all experienced situations where a long sequence of e-mails was needed to be exchanged to set a date. Clearly, channel properties and performative types are closely related, i.e., for a given performative certain properties are appropriate while others are not. For example, for the performative "You are fired!" a point-to-point, reliable, secure, synchronous, closed, and formal channel is most appropriate. For the performative "Happy bithday!" a point-to-point, synchronous, and informal channel is more appropriate. In the latter case, reliability, security, and closure are of less importance.

3.4. Naming service

All interaction is based on proclet identifiers (proc_id's) and class identifiers (class_-id's). These identifiers provide the handles to route performatives. By sending a performative using a class_id, all instances of the corresponding class receive the performative. Only if a proclet knows the proc_id's of the recipients of the performative, it is able to communicate with specific proclets. In many situations the sending proclet does not know the proc_id's of all receiving proclets. Therefore, we introduce the concept of the *naming service*. The naming service keeps track of all proclets and can be queried to obtain proc_id's. There are many ways to implement such a naming service. Consider for example the services provided by the object request brokers developed in the context of CORBA. In this paper, we only consider the desired functionality and abstract from implementation details (e.g., distribution of the naming service over multiple domains).

The naming service provides the following primitives: *register*, *parent*, *child*, *update*, *unregister*, *query*, and *forward*.

The function *register* is called by the proclet the moment it is created. Therefore, the execution of one of the create tasks (i.e., source transitions) coincides with the execution of the *register* primitive. The primitive has the following parameters: *creator* (i.e., the proc_id of the calling proclet), *time* (i.e., the time the function is called), *class name* (i.e., the name of class of the created instance), *owner* (i.e., the identity of the actor responsible for the proclet) and *attributes* (i.e., the characteristic properties of the created proclet) and returns a new unique proc_id. The proc_id is returned by the naming service in order for the proclet to know its own identity. Proclets can be created by other proclets. Consider for example Figure 3. The create task *Create entry* is triggered by a performative sent by a *Meeting* proclet. The performative is created by the task *Invite for meeting*. This implies that the task *Invite for meeting* already registered the new *Personal entry* proclet. The new proclet is already registered by the meeting proclet because the meeting proclet needs a handle to the newly created proclet. Since proclets can be created by other proclets, there are parent-child relationships. The functions *parent* and *child* can be used to navigate though the naming service. Both functions have a proc_id parameter. The *parent* function returns a proc_id (if any) and the *child* function returns a set of proc_id's.

The proclet attributes registered in the naming service describe the essential characteristics, e.g., role and group attributes, links to actors, etc. The set of attributes is not fixed and may vary from one class to another. During the life cycle of the proclet these attributes may change. The function *update* with parameters *proc_id* and *attributes* can be used to change existing or add new attributes.

Based on the attributes, proclets can query the naming service using the function *query*. The function has one parameter describing a Boolean expression in terms of attributes and returns a set of proc_id's, i.e., all proclets satisfying the expression.

Entries in the naming service can be removed using the function *unregister*. Executing a finish task (i.e., a sink transition in the WF-net) results in a call to *unregister*. Function *unregister* has one parameter: The proc_id of the proclet to be destroyed.

Sometimes there is a need to merge proclets. Consider for example two proclets corresponding to the same traffic accident. If two police officers file a report on the same traffic accident, two proclets are created. If after executing some steps it turns out that both proclets correspond to the same traffic accident, then it does not make sense to execute the remaining tasks for both proclets. Therefore, we propose to merge the two proclets by destroying one of them and redirecting all performatives to the remaining one. For this purpose we propose the function *forward*. This function has two proc_id parameters: one for the destroyed proclet and one for the remaining proclet. As a result of calling this function, all performatives intended for the destroyed proclet are redirected to the remaining proclet.

3.5. Actors

Proclets have *owners*. Owners are the actors responsible for the proclet. Actors can be automated components, persons, organizations (e.g., shipping department), or even whole companies. Owners are specified at proclet registration time and this information is kept by the naming service (see Section 3.4). Ownership can be transferred by updating the naming service information.

The owner will sometimes be the executor of proclet tasks him or herself - in the example of Figure 3, for instance, the owner of the personal entry will most probably be the one that will perform the tasks, essentially the decision of attending or skipping the meeting. Roles may be specified for each task, in which case the executor can be different from the owner. We assume that the usual role resolution mechanisms[25] are employed in this latter case.

We propose to model as *external proclets* those actors (in the broad sense of the word) that interact with proclets in a more complex way. External proclets are useful to model those interactions that go beyond the simple model assumed by the usual role mechanism, e.g., when a request for service may be either accepted, rejected or counter-proposed. External proclets, as the name implies, represent entities that are outside of the scope of the workflow process, whereas *internal proclets* are those under the control of the workflow system enactment service. Both types of proclets are modeled in a similar way - by describing expected interactions with other proclets. Note that external proclets are not instantiated by the workflow management system. The entities they represent exist independent of the workflow considered. Detailed examples of both *internal* and *external proclets* are presented in Section 4.

4. Example Revisited

We now revisit the conference paper selection process, this time using proclets. The multiple perspectives of conference, paper and review that were identified in Section 2 as being one of the obstacles for representation are taken into account and integrated into a seamless model. The resulting model has a much broader scope than the ones usually found in the literature. In particular, interactions with the environment are made explicit.

The model is composed of six proclets, with well defined interfaces, that correspond to the class diagram entities previously presented (Figure 1). Three of the proclets correspond to *internal proclets* (Figures 6, 7, 8) and the other three are *external* (Figures 9, 10, 11).

The *Conference* proclet groups tasks that act upon or require access to the set of all submitted papers, e.g., the distribution among PC members and final selection of papers. For each *Conference* proclet, there will exist many related *Paper* proclets - one instance per paper. Each *Paper* proclet will in turn be associated to some *Review* proclets. There will be as many *Review* proclets as there are reviews. The multiple instances of *Paper* and *Review* proclets directly reflect the multiple cardinality of the relationships between conference, paper and review as shown in the class diagram (Figure 1). *Author*, *PC member* and *Reviewer* are external proclets and specify the details of the interactions between these actors and the internal proclets.

We now analyze in more detail the *Conference* proclet (Figure 6). The first few tasks in this proclet deal with the staffing of the program committee (PC). *Invite PC member* sends out a multicast message to prospective PC members. These invitations will either be accepted or rejected. In case of rejection, a new round of invitations can take place. Note that here the responses to the illocutionary act *invite* are explicitly included in the model. The single multicasted invitation will be responded to asynchronously by the persons that were invited, so the tasks *Accepted* and *Rejected* are enabled in a loop and receive multiple messages, one at a time, as indicated by the cardinality 1 and multiplicity $*$ of the associated ports.

The task *Replace rejected* should obviously only fire if one or more *rejection* performatives were received. This part of the model therefore illustrates the need and use of knowledge bases. *Replace rejection* has a pre-condition that queries the knowledge base and only allows firing if at least one *rejection* has been received. Similarly, as soon as a certain number of PC members have accepted the invitation, the pre-condition for the task *Call for papers* will enable it to fire.

After the committee is staffed, a call for papers is issued, multicasted to many prospective authors. In practice the recipients of this multicast performative will be mailing lists and individuals whose identities are stored in some database. Once

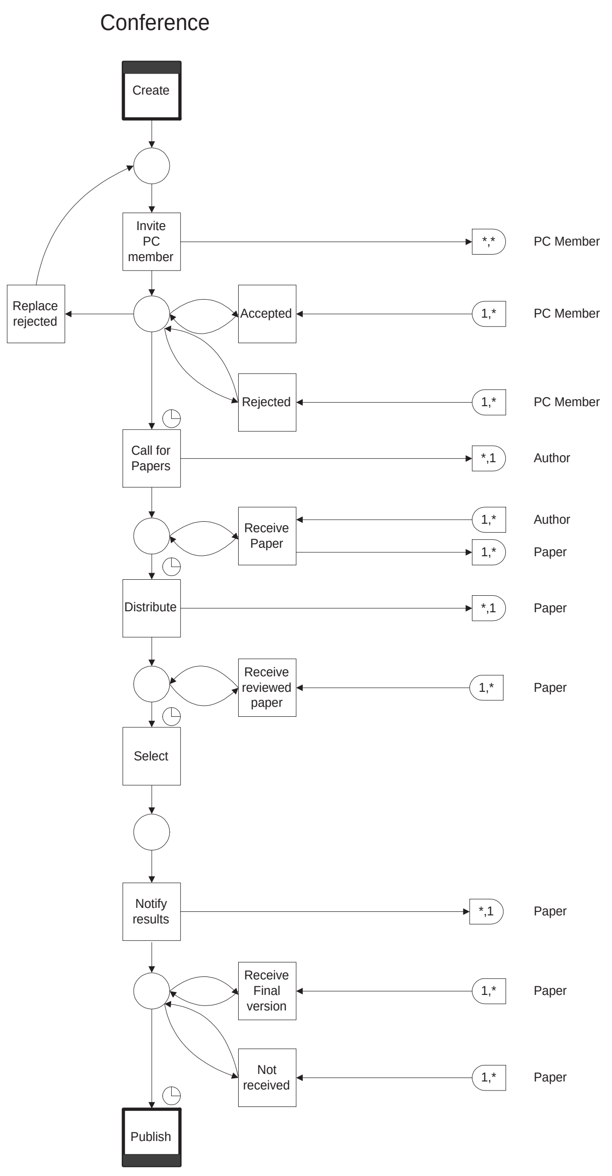


Figure 6: Conference proclet.

again, the responses will be received one by one, in separate asynchronously generated messages. *Receive paper* therefore is enabled in a loop that receives the submissions and that sends a performative that creates new instances of *Paper* proclets. The result is that there will eventually exist as many *Paper* proclet instances as there are submissions.

The *Distribution* task of the *Conference* proclet corresponds to the decision making as to which paper should be handled by which PC members. Once this decision is made, a performative informs each *Paper* proclet (Figure 7) about the identity of assigned PC members. The *Paper* proclet, in turn, generates a second multicast, this time to create as many *Review* proclets (Figure 8) as needed (one for each assigned PC member). The performative to each *Review* proclet informs about the identity of the responsible PC member. This illustrates one basic design principle - work is distributed through the proclets in such a way that each proclet deals only with tasks that are at the same aggregation level. The *Conference* proclet, for instance, groups tasks that operate on the whole set of submitted papers, while *Paper* proclets handle work at the individual paper level. *Review* proclets group tasks that pertain to each of the multiple individual reviews a paper has.

Back in the *Conference* proclet, *Selection* decides whether papers should be accepted or rejected based on their relative merits. The final decision is multicasted to the Paper proclets, that notify the authors and then wait for the reception of final versions of those papers that were accepted.

The tasks in the remaining part of the *Conference* proclet collect the final versions of the papers and deal with problem reports originating from the *Paper* proclet. *Publish* is the final step in the proclet.

To make the communication between the internal proclets and the environment explicit, we model authors, pc members, and reviewers as *external proclets*. Each of the corresponding proclets describes the "state of mind" of the respective actors with respect to the conference at some point in time. *External proclets* are lightweight in the sense that they do not imply that the environment conduct business in that specific manner, only that it is compatible with the specified communication behavior. Typically, *external proclets* do not correspond to an executing object, and usually just reflect the fact that in the environment there is an actor that can be expected to behave according to some communication protocol. *External proclets* allow us to make these assumptions explicit, making them visible and verifiable, through inspection and/or simulation.

Initially, authors receive the call for papers (or hear about it from a friend), submit papers (or not), receive acknowledgments and provide requested information (if any) until the submission deadline. These possible interactions are modeled by the *Author* proclet (Figure 9). From the point of view of this proclet, there are no explicit constraints on the order in which these messages will be generated/received. Note that one author can submit multiple papers for the same conference. Therefore, acknowledgments, submissions, etc., can be interleaved.

In a similar way, *PC member* and *Reviewer* proclets (Figures 10 and 11) model

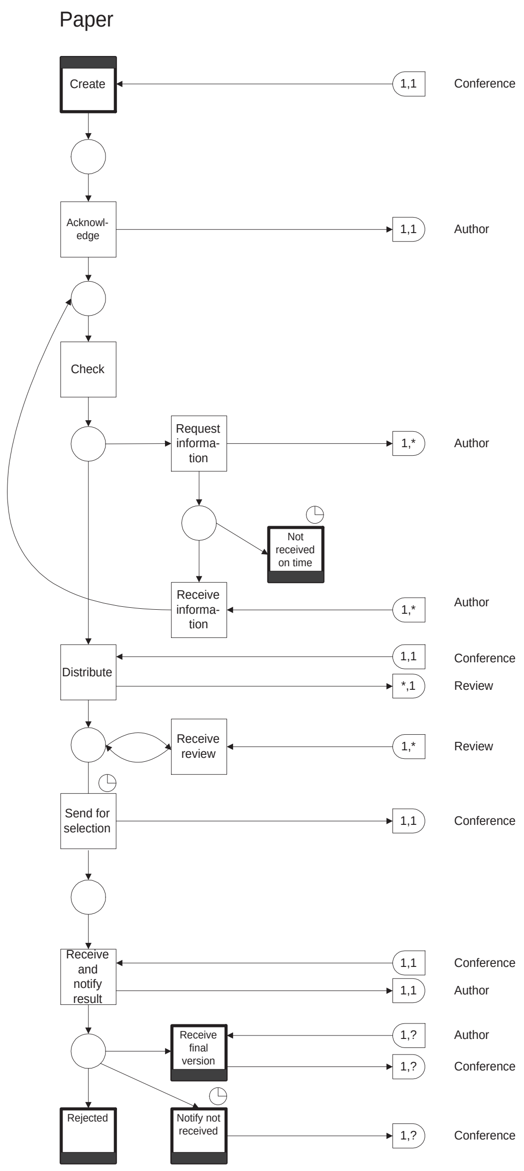


Figure 7: Paper proclets.

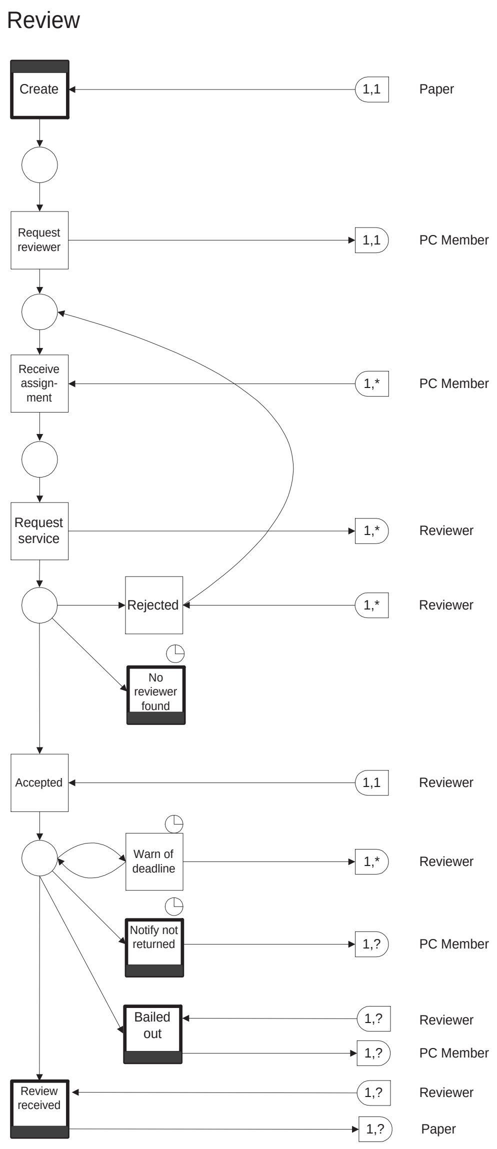


Figure 8: Review proclet.

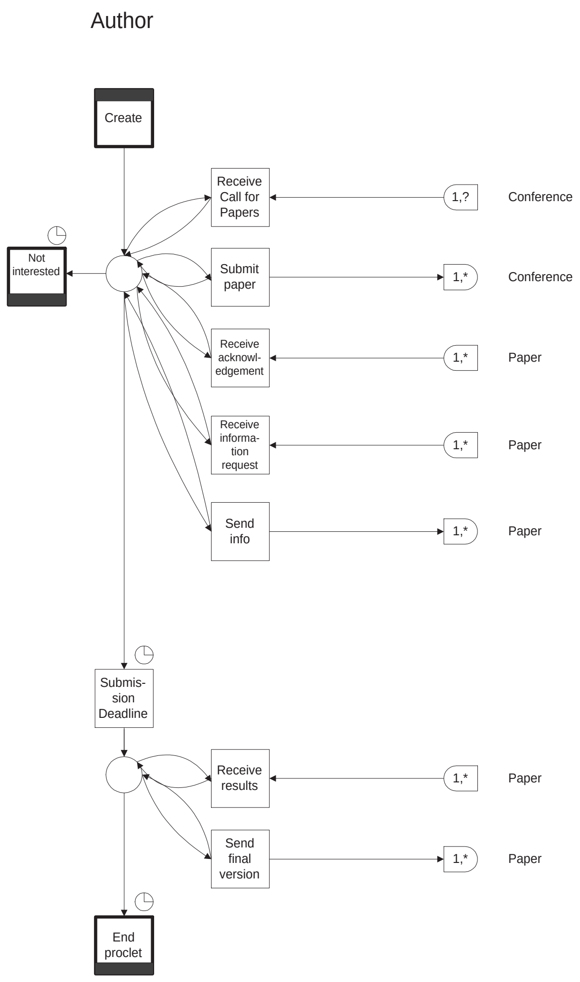


Figure 9: Author proclet.

expected interactions. An important difference is that explicit responses from these actors are expected, specifically regarding the invitations to join the process. While authors will not typically inform the PC that they are not interested in submitting papers, acceptance or rejection of invitations on the part of prospective PC members and reviewers have a direct impact on the process - acceptance implies commitment to perform required work, and rejection causes actions to find replacements.

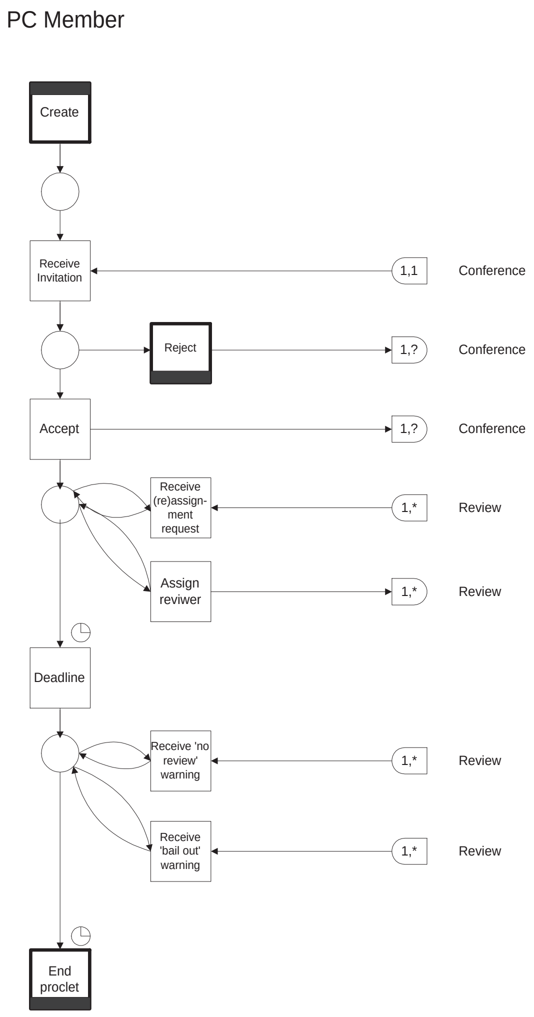


Figure 10: PC member proclet.

Another aspect worth examining is the way by which reviewers are invited to review a paper. Each of the multiple instances of the *Review* proclet will execute task *Request reviewer*, asking the responsible PC member to assign a reviewer. After an assignment is received, the *Review* proclet requests service from this prospective

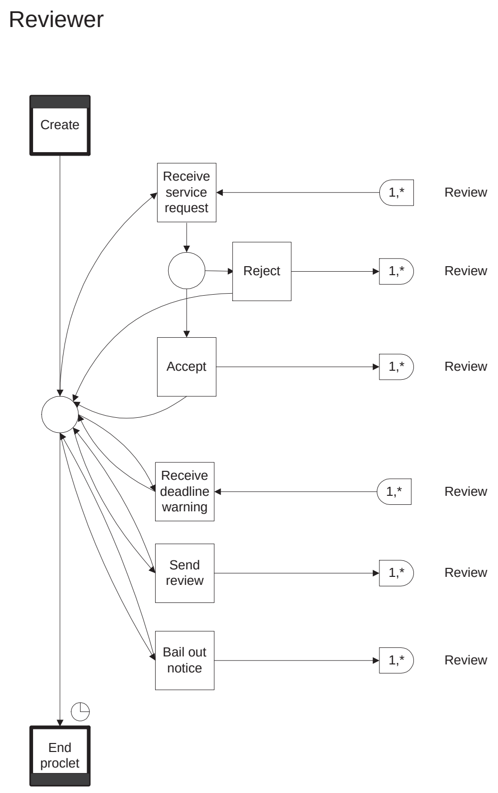


Figure 11: Reviewer proclet.

reviewer, by sending a *request* performative. In case of rejection, the proclet itself manages the request for a replacement. These steps are repeated until either a willing reviewer is found or time runs out.

Note that the model presented here includes aspects that cannot be represented by other modeling languages. In particular:

- The different perspectives, corresponding to the three different level of aggregation, conference, paper, and review, that were identified in the class diagram (Figure 1) are explicitly represented.
- The transition between these different levels of aggregation are cleanly specified as communication between proclets.
- Launching of variable number of instances of lower aggregation elements, and their synchronization - grouping and ungrouping - can be easily and clearly represented.

As motivated in the introduction, traditional workflow management systems are unable to deal with these issues. As a result, the workflow process is changed to accommodate the workflow management system, the control-flow of several cases is artificially squeezed into one process definition, or the coordination amongst cases is hidden inside custom built applications. These unsatisfactory "patches" can be avoided by adopting the framework presented in this paper. The framework also encourages broadening the scope of what is represented, making explicit some of the usually hidden assumptions:

- *External proclets* can be used to represent actors that are part of the environment. These are typically omitted from models, which makes them harder to verify.
- *Performatives* offer a mechanism to more precisely model message content. Speech-act theory can be used to clarify and regulate the semantics of interactions.
- *Channels* offer a way to explicitly represent (and eventually support at enactment) different media and their attributes.

It is also important to realize that, even though much more is represented, the resulting model is composed of small modules with clear-cut interfaces to the environment. Furthermore, these modules have a one-to-one correspondence with the entities of the class diagram, which can, therefore, be used as a guideline for proclet development. This is usually not the case in existing modeling languages: models often are monolithic and large; there is usually no close connection between the resulting models and the problem space, as mapped, for instance, by a class diagram. All these features come in addition to the full expressive power of Petri nets, a formalism that has proven to be especially adequate for representing processes in general and workflows in particular.

5. Another Example: Hiring New Employees

The example in the previous section illustrated the use of proclets to deal with multiple levels of granularity/aggregation (conference, paper, and review). In this section, we consider an example which is not characterized by multiple levels of aggregation but by two complementary aspects. The example shows a possible hiring process, that considers both *advertised* and *non-advertised* job applications:

(1) *Advertised applications*: hiring is normally started by the creation of a new position to be filled in a department. New positions are advertised and candidates send in applications that are evaluated and lead to a final selection. Typically, multiple candidates apply for one position.

(2) *Non-advertised applications*: it is also possible that candidates will send in applications even if no position has been advertised. Depending on the qualifications of a candidate and interest of some department, a new position can be created just so that this candidate can be hired.

Again, multiple perspectives can be identified in such a process. One can choose either *position*, or *application* as the central concept, but either choice causes representation problems, due to the fact that, again, some tasks need to consider the whole set of applications for a position (e.g., final selection of the candidate that will fill the position), while others consider each individual application (e.g., the screening of each application).

Using proclets, both perspectives can be represented, along with external proclets that make clear the roles of candidates, departments and Personnel. Therefore, the model involves five proclet classes: (1) *Position*, (2) *Application*, (3) *Department*, (4) *Candidate* and (5) *Personnel*. Figures 12 through 16 display the five proclets. Transitions are colored with three shades of gray to show the occasions in which they might be enabled:

- *White* transitions can be enabled both when *advertised* and *non-advertised* applications are being processed;
- *Light gray* transitions correspond to ones that deal with the situation where a position is *advertised*;
- *Darker gray* marks transitions that are used to handle *non-advertised* applications.

These colors have no semantic significance. Colors are there for the sole benefit of the human readers. As always, tasks are enabled if (a) the corresponding transition in the WF-net is enabled, (b) the precondition evaluates to true, and (c) each input port contains a performative (see Section 3.1).

We start by analyzing the situation where a position is *advertised*. In terms of diagram colors, we will examine those parts that are white and light gray. The first step in the processing of an advertised position is taken by a department, which creates a new position. *Create Position*, in the *Department* proclet (cf. Figure 12), is responsible for that. The performative generated by this transition causes the creation of a new instance of a *Position* proclet (cf. Figure 13).

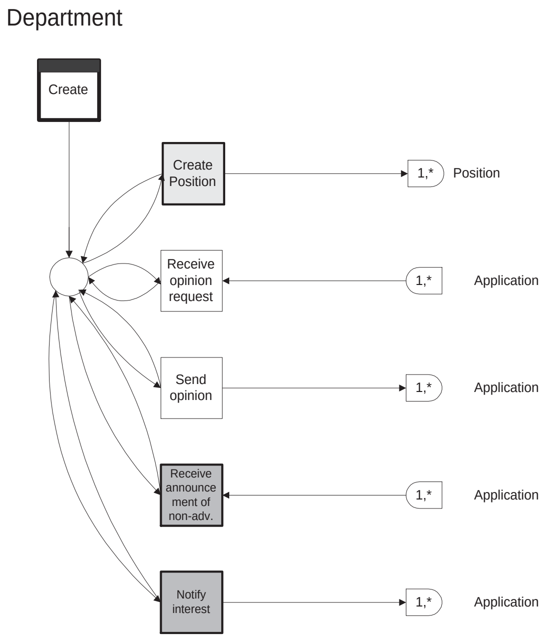


Figure 12: Department proclet.

Necessary processing for a new position is handled by the *Position* proclet that, through interaction with other proclets, coordinates the process of filling the position. The first step consists of a multicast that advertises the new position to potential candidates (*Advertise* in Figure 13). Similar to the conference example, it is assumed that there is some mechanism to inform potential candidates, i.e., we assume that for each potential candidate there is a corresponding proclet and the performative sent by the *Position* proclet is received by each of these proclets. This assumption is not very realistic. However, since the *Candidate* proclet is external and not really a part of the workflow process being considered this abstraction is acceptable. Note that we can model the process of triggering candidates in more detail to get a more accurate model, e.g., we can add newspaper proclets and people subscribing to these newspaper proclets. However, for this example it seems reasonable to use the abstraction presented.

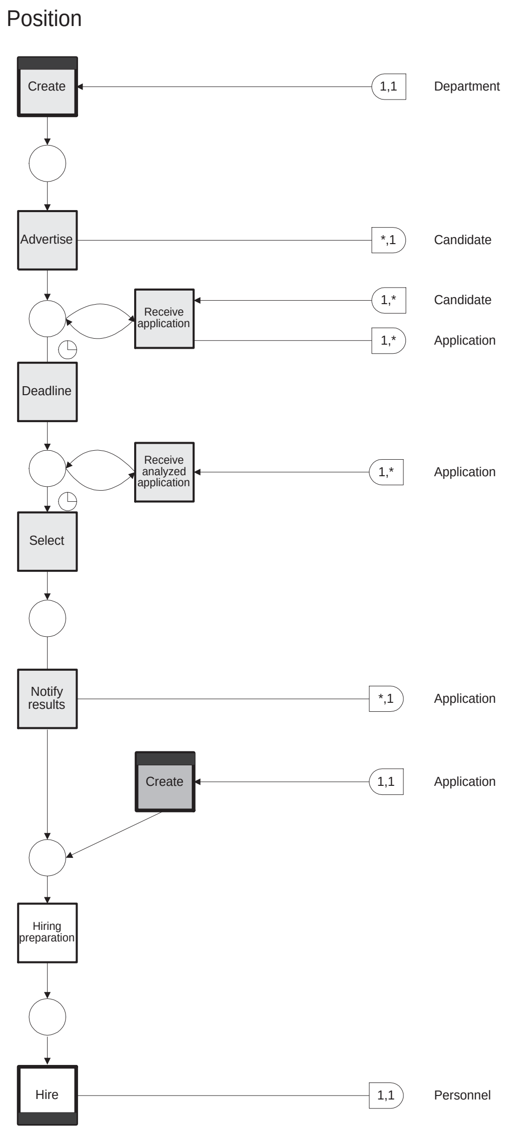


Figure 13: Position proclet.

Applications from interested parties are received in a loop following the advertisement. For each application that is received, a performative that creates an *Application* proclet is generated. As we can see, the pattern here is similar to the one employed in the *Conference* proclet (Figure 6): there, paper submissions were received in a similar loop that followed a multicasted call-for-papers. After a deadline is reached, reception of the analysis of each application is enabled. The analysis itself is performed in the *Application* proclet as we will examine momentarily. Once the analysis of all applications have been received or a deadline has been reached, the selection takes place. Similarly to what happens in the *Conference* proclet with respect to the papers, selection of a candidate to fill a position demands that all available applications are considered as a whole.

Once selection has been concluded, *Position* multicasts a performative back to the multiple *Application* proclets, to have them notify the results (approval or not) of each candidate. Finally, after some preparation, a performative requesting hiring of the selected candidate is sent to *Personnel*, and the proclet finishes. In this simple model, we assume a suitable candidate will always be chosen. The proclet can be easily adapted to handle the case where all candidates might fail.

Having examined how processing at the *Position* level takes place, we now turn our attention to the *Application* proclet (Figure 14), which deals with the bulk of the processing, namely, that of each individual application for a position. The *Application* proclet has two *Create* transitions. Recall that under the single activation assumption (Section 3.1), only one of these will fire (once) for each proclet instance. We are concerned at the moment with the *Create* that is enabled as a response to a performative originating from *Position* (the left one in the diagram). The other *Create* has its origin in a *Personnel* proclet and corresponds to the processing of a *non-advertised* application, which we will analyze later in the text.

After being created as a response to a performative sent by *Position*, each advertised application is analyzed according to different phases that we describe generically as *Phase 1*, *Phase 2* and so on. The details of such phases may vary depending on the organization policy regarding selection and hiring. Each phase can cause the application to be immediately rejected, in which case all others are skipped. For illustrative purposes, we included a phase where departments are consulted regarding the adequacy of an application. One or more departments can be requested to review an application and send back an opinion. This request is sent by *Consult departments* and corresponds to a multicasted performative (as indicated by the $*$ cardinality). Given that an early rejection might take place before the proclet ever reaches this state, the multiplicity is ?, indicating that zero or one performatives will be generated during the life-cycle of this proclet.

After an interview takes place (or not), the result of the analysis is sent back to the *Position* proclet were, as we already examined, all applications for a position are considered and selection is made. The result of this selection is received back at the *Application* proclet, and candidates are notified of the outcome.

Candidate proclets (Figure 15) are *external proclets* that represent interactions

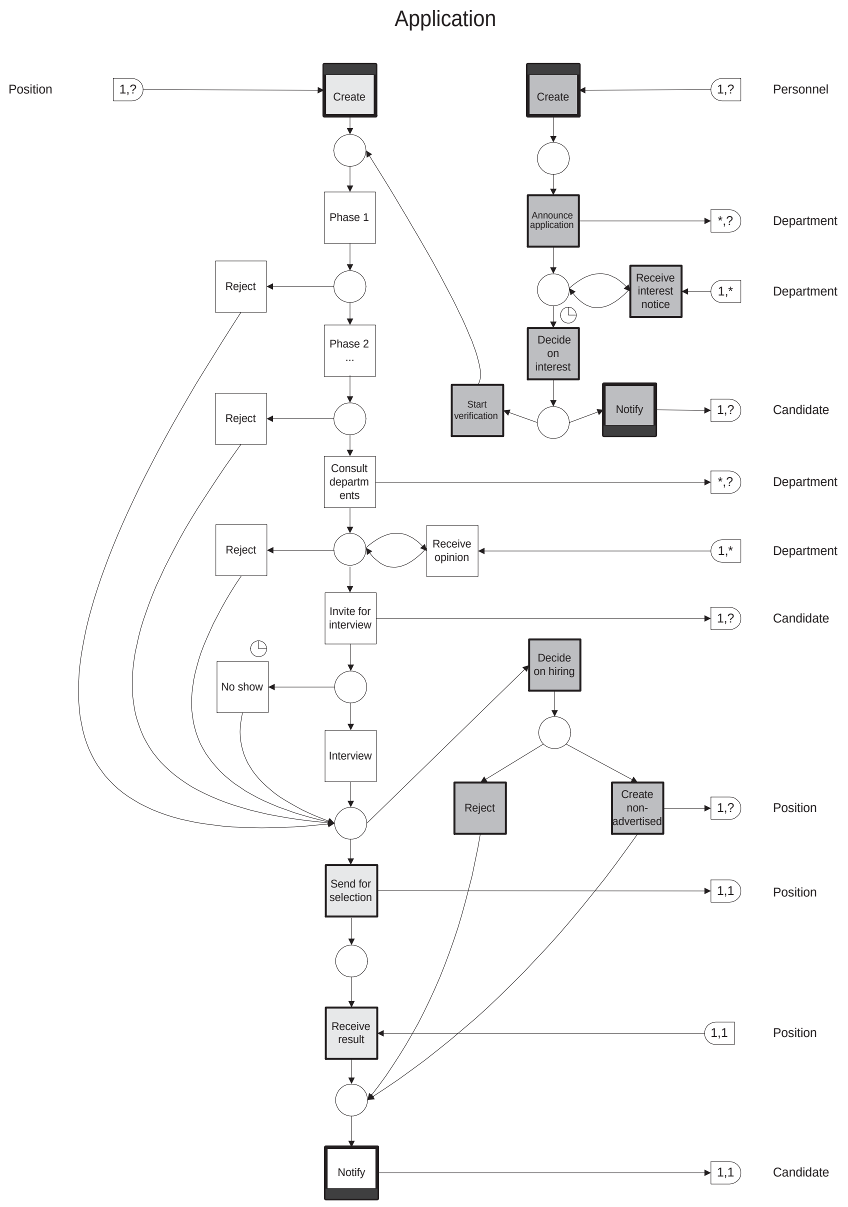


Figure 14: Application proclet.

with candidates. Candidates receive advertisements and either do not respond, if not interested, or submit applications. In the case of *advertised* positions, *Position* is the recipient of a performative containing the application sent by a candidate, in which case processing follows the sequence that we described so far. As mentioned before, candidates can also submit applications even if a position has not been advertised. We now describe how such *non-advertised* applications are dealt with by the proclets (in transitions that are painted dark gray).

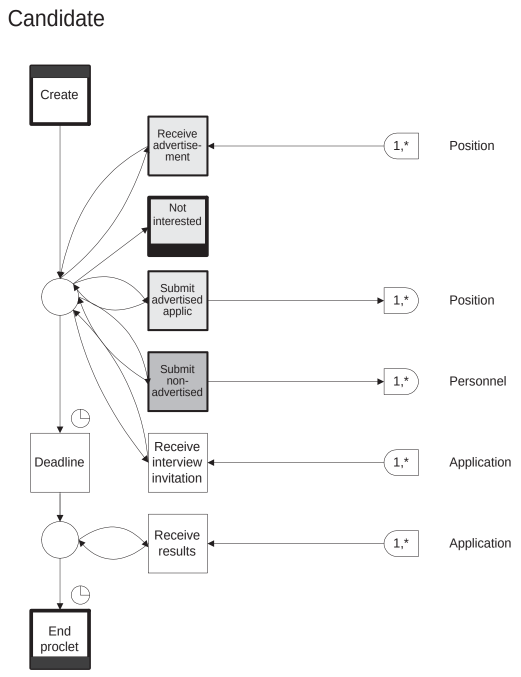


Figure 15: Candidate proclet.

Submission of *non-advertised* applications are received by a *Personnel* proclet (Figure 16). This external proclet describes interactions of this business unit with respect to hiring. Besides receiving requests for hiring, *Personnel* is also responsible for handling the initiation of processing in case of *non-advertised* applications. This processing consists of generating a performative that causes an *Application* proclet instance to be created. Note that unlike the *advertised* situation we examined so far, this *Application* proclet instance is not related to any *Position* proclet yet. This shows the flexibility of the proposed framework, that can handle very different initiation paths with just a few extra and alternative steps added to the proclets already examined (those painted dark gray in the diagrams).

Recall that *Application* (Figure 14) has two distinct *Create* transitions. We now analyze the sequence that takes place when the creation has its origin as a response to a performative generated by *Personnel*, that correspond to those transitions painted dark gray.

The first step after creation of a *non-advertised* application as a response to a performative sent by *Personnel* is to announce internally the arrival of a new application. Since there is not a specific department opening a position, this announcement is broadcast to departments and a loop collects responses. In case one or more departments show interest in the application, verification is started, following the usual steps that were examined in the context of *advertised* positions.

Processing of a *non-advertised* application takes a different route right after the interview. Instead of sending for selection, which only makes sense in cases where a *Position* has been advertised (and therefore already exists as a proclet), the decision on hiring takes place in the *Application* proclet itself. Such decision can be reached in the context of the *Application*, given that there are no competing candidates that need also to be considered. If the application is considered to be acceptable, *Create non-advertised* sends a performative that causes the creation of a new *Position*. Additional processing in the *Position* proclet (Figure 13) consists just of this extra creation transition, that shortcuts processing, including only the final steps that concern hiring of this (approved) candidate.

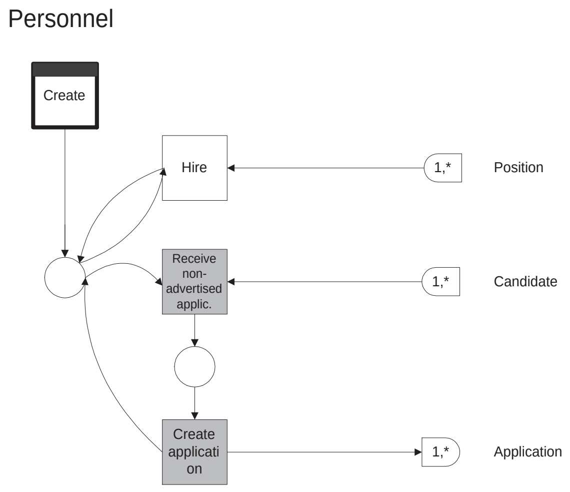


Figure 16: Personnel proclet.

In this section, we showed that proclets can be used to represent a workflow process which can handle two different types of cases: *advertised* and *non-advertised* positions. The resulting model is quite natural and reuses as much as possible, i.e., due to the proclet structure there is no duplication of process parts.

The reader familiar with the work on WF-nets[17] will have noticed that the department proclet (Figure 12) and the personnel proclet (Figure 16) do not correspond to a WF-net in a technical sense. These proclets are considered persistent and therefore no finishing tasks (i.e., sink transitions) have been added. These tasks can be added to obtain a WF-net. Apart for this technicality, each proclet class presented in this paper corresponds to a WF-net. Moreover, each of these WF-nets is sound (see Section 3.1). In fact these diagrams have been verified using our workflow verifier Woflan.[21] The ability to verify these nets using Woflan illustrates the added value of having Petri-nets as a starting point.

6. Related work

Petri nets have been proposed for modeling workflow process definitions long before the term "workflow management" was coined and workflow management systems became readily available. Consider for example the work on Information Control Nets, a variant of the classical Petri nets, in the late seventies.[19,20] Since then many workflow models and languages have been developed ranging from approaches based on other formal models such as state charts[26] to the vendor-specific diagramming techniques used in the many commercial workflow management systems available today. Workflow models described in the literature focus on various aspects[2] such as transactional concepts[27], flexibility[28], analysis[17], and cross-organizational workflows[29,30], etc. Any attempt to give a complete overview of these models is destined to fail. Therefore, we only acknowledge the work that extended workflow models to accommodate the problems identified in Section 2.

Zisman presents a paper refereeing example that involves Petri-nets and allows multiple instantiation of the reviewer net.[31]

In workflow literature, many authors have observed the problems related to multiple instances of a task.[32,33,34,35] The concept of a so-called batch-oriented task[32] has been proposed to allow for a task that is executed for multiple instances at the same time. To support batch-oriented tasks, independent cases need to be synchronized. As an example, consider the task of selecting papers for a conference (task *select* in the example): All papers are considered at the same time. The need to deal with the batch-oriented clustering of instances was also recognized in the Wide model.[33] A similar extension is proposed by Casati et al. using so-called *multi-tasks*.[34] A multi-task is a task in a process which can be instantiated an arbitrary number of times. The multi-task completes the moment that the corresponding task is completed for each instance, or for some number of those instances (the quorum). A similar mechanism has been implemented for the Regatta system by Fujitsu.[36] In this system, multiple instances are created, according to the number of actors available to perform them. In Spade-1, a process-centered software engineering environment (PCSEE), it is possible to instantiate dynamically the same activity a variable number of times, generating different execution threads for the activity, called *active copies*.[37] Other workflow languages supporting similar constructs are IBM's FlowMark (the bundle it is no longer available in MQSeries/Workflow) and FLOWer (through the so-called dynamic subplan).

The idea to promote interactions to first-class citizens was proposed in different settings. For example, in the context of the language/action perspective[12,13,14,15], Action Technologies developed a workflow tool[38] where each step in the process is characterized by four phases: preparation, negotiation, performance and acceptance. The transition from one phase to another is mainly driven by interactions between actors. In the more systems-oriented domains there have also been some proposals for inter-process communication. Consider for example Opera[39], a process support system kernel (i.e., a rudimentary workflow management system), which supports the interaction between different processes.

The language-action perspective is also employed in the context of agent technology.[13] Speech-acts form the basis for performatives in agent interaction languages, e.g. KQML.[24] The use of agents for implementing workflow systems is explored, e.g., in the Bond multi-agent system.[40] Petri-nets are used in Bond as an intermediate representation of workflows.[41] Another example of the application of agents to workflow is the the so-called "worklets" model and architecture.[42] Worklets are scripted mobile workflow agents that can "hop" from one component to another. Worklets are similar to proclets in the sense that they are lightweight workflow processes. However, the emphasis is on mobility rather than interaction.

Another line of research related to the results presented in this paper is the work on so-called workflow patterns.[4] The workflow patterns home page[5] also includes patterns for dealing with multiple instances, i.e., multiple levels of granularity/aggregation. This work extends the work on workflow patterns.[35] Some of the ideas presented in the related work mentioned in this section have been adopted by our framework: batch-oriented operation, multi-tasks, and inter-process communication can be handled easily by the framework. In addition, the framework employs concepts such as performatives, channels, ports, knowledge bases, naming services, and the rigor of a Petri-net basis which allows for various forms of analysis and a straightforward and efficient implementation.

7. Conclusion

In this paper, we presented a framework which advocates the use of interacting proclets, i.e., lightweight workflow processes communicating by exchanging performatives through channels. As was demonstrated in this paper, the framework can solve many of the traditional modeling problems resulting from the case-oriented paradigm.

If we compare the proclet framework with traditional workflow languages, the following differences can be observed.

- A workflow model in terms of proclets is closer to reality since it is not necessary to squeeze the description into a single process definition.
- There is a natural link between class diagrams presenting a static view on the workflow and the proclets representing a dynamic view on the workflow.
- Proclets are more expressive than traditional workflow languages, e.g., problems related to multiple instances can be resolved without resorting to coding.

- Multiple workflows may share proclets. Therefore, reuse is supported and duplication of process information is avoided.

It should be noted that a workflow model in terms of proclets is of about the same size as a traditional workflow model if we define the size as the *total* number of nodes/tasks. However, the size of one proclet class is typically much smaller that the overall workflow process. This allows for a "divide and conquer" approach at the level of workflow processes.

In the future, we plan to explore the relation between channels and performatives. We are also compiling a list of interaction patterns. In our view, the interaction between proclets typically follows a number of well-defined patterns, e.g., a request performative is followed by an accept or reject performative. For structuring these patterns we can use the notion of inheritance of dynamic behavior.[43] Finally, we plan to build a prototype to support the framework. One idea is to provide a proof of concept by experimenting with ExSpect.[44] ExSpect is a Petri-net-based prototyping environment which can be used to simulate workflows.

References

1. P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition.* John Wiley and Sons, New York, 1997.
2. A.P. Sheth, W.M.P. van der Aalst, and I.B. Arpinar. Processes Driving the Networked Economy: ProcessPortals, ProcessVortex, and Dynamically Trading Processes. *IEEE Concurrency*, 7(3):18–31, 1999.
3. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation.* International Thomson Computer Press, London, UK, 1996.
4. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, Berlin, 2000.
5. Workflow Patterns Home Page. http://www.tm.tue.nl/it/research/patterns/.
6. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1998.
7. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets II: Applications*, volume 1492 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1998.
8. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, Reading, MA, USA, 1998.
9. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley, Reading, MA, USA, 1998.
10. R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.
11. N. Jennings and M. Wooldridge, editors. *Agent Technology : Foundations, Applications, and Markets.* Springer-Verlag, Berlin, 1998.
12. F. Flores and J.J. Ludlow. Doing and Speaking in the Office. In *Decision Support Systems: Issues and Challenges*, pages 95–118. Pergamon Press, New York, 1980.

13. E.M. Verharen, F. Dignum, and S. Bos. Implementation of a cooperative agent architecture based on the language-action perspective. In *Intelligent Agents*, volume 1365 of *Lecture Notes in Artificial Intelligence*, pages 31–44. Springer-Verlag, Berlin, 1998.
14. T. Winograd. Special Issue on the Language Action Perspective - Introduction. *ACM Transations on Office Information Systems*, 6(2):83–86, 1988.
15. T. Winograd and F. Flores. *Understanding Computers and Cognition: A New Foundation for Design*. Ablex, Norwood, 1986.
16. A. LaMarca, W.K. Edwards, P. Dourish, J. Lamping, I. Smith, and J. Thornton. Taking the Work Out of Workflow: Mechanisms for Document-Centered Collaboration. In *Proceedings of the Sixth European Conference on Computer-Supported Cooperative Work (ECSCW'99)*, 1999.
17. 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.
18. N.R. Adam, V. Atluri, and W. Huang. Modeling and Analysis of Workflows using Petri Nets. *Journal of Intelligent Information Systems*, 10(2):131–158, 1998.
19. C.A. Ellis. Information Control Nets: A Mathematical Model of Office Information Flow. In *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 225–240, Boulder, Colorado, 1979. ACM Press.
20. 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.
21. H.M.W. Verbeek and W.M.P. van der Aalst. Woflan 2.0: A Petri-net-based Workflow Diagnosis Tool. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 475–484. Springer-Verlag, Berlin, 2000.
22. Woflan Home Page. http://www.tm.tue.nl/it/woflan.
23. J.R. Searle. *Speech Acts*. Cambridge University Press, Cambridge, 1969.
24. T. Finin, J. Weber, G. Wiederhold, and et. al. Specification of the KQML Agent-Communication Language , 1993.
25. M. zur Mühlen. Evaluation of workflow management systems using meta models. In *Proceedings of the 32nd Hawaii International Conference on System Sciences - HICSS'99*, pages 1–11, 1999. http://www.computer.org/proceedings/Hiccs2/0001/00010198Babs.htm.
26. P. Muth, D. Wodtke, J. Weissenfels, A. Dittrich, and G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems*, 10(2), 1998.
27. 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.
28. M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
29. 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.
30. W.M.P. van der Aalst. Process-oriented Architectures for Electronic Commerce and Interorganizational Workflow. *Information Systems*, 24(8):639–671, 2000.
31. M. D. Zisman. Use of production systems for modeling asynchronous concurrent processes. *Pattern-Directed Inference Systems*, pages 53–68, 1978.
32. P. Barthelmess and J. Wainer. Workflow systems: a few definitions and a few sug-

gestions. In N. Comstock and C.A. Ellis, editors, *Proceedings of the Conference on Organizational Computing Systems - COOCS'95*, pages 138–147, Milpitas, California, September 1995. ACM Press.

33. F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Snchez. Wide workflow model and architecture. Technical report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1996. http://dis.sema.es/projects/WIDE/Documents/ase30_4.ps.gz.
34. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. In *Proceedings of the OOER International Conference*, Gold Cost, Australia, 1995.
35. W.M.P. van der Aalst, P. Barthelmess, C.A. Ellis, and J. Wainer. Workflow Modeling using Proclets. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 198–209. Springer-Verlag, Berlin, 2000.
36. K. Swenson. Collaborative planning: Empowering the user in a process environment. *Collaborative Computing*, 1(1), 1994. ftp://ftp.ossi.com/pub/regatta/JournalCC.ps.
37. S. Bandinelli, M. Braga, A. Fuggetta, and L. Lavazza. Cooperation support in the spade environment: a case study. In *Proceedings of the Workshop on Computer Supported Cooperative Work, Petri nets, and Related Formalisms (14th International Conference on Application and Theory of Petri Nets)*, Chicago, June 1993. ftp://ftp-se.elet.polimi.it/dist/Papers/ProcessModeling/CSCWPN93.ps.
38. Action Technologies. *ActionWorkflow Enterprise Series 3.0 User Guide*. Action Technologies, Inc., Alameda, 1996.
39. C. Hagen and G. Alonso. Beyond the black box: Event-based inter-process communication in process support systems (extended version). Technical report, ETH Zürich, July 1997. Technical Report No. 303. http://www.inf.ethz.ch/department/IS/iks/publications/files/ha98c.pdf.
40. Purdue University. Bond. the distributed object multi-agent system. http://bond.cs.purdue.edu, 2000.
41. K. Palacz and D.C. Marinescu. An agent-based workflow management system. In *Proc. AAAI Spring Symposium Workshop "Bringing Knowledge to Business Processes"*, Standford University, CA, March 1999. http://bond.cs.purdue.edu/docs/papers/awfms.ps.
42. G. Valetto, G.E. Kaiser, and G.A. Kc. A Mobile Agent Approach to Process-Based Dynamic Adaptation of Complex Software Systems. In V. Ambriola, editor, *Proceedings of the 8th European Workshop on Software Process Technology*, volume 2077 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, Berlin, 2001.
43. W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An approach to tackling problems related to change. *Theoretical Computer Science*, 2001 (to appear).
44. Deloitte & Touche Bakkenist. ExSpect Home Page. http://www.exspect.com.