# A Model for Extensible Web Based Information Intensive Task Oriented Systems

*Cecília Kremer Vieira da Cunha*[1] *Clarisse Sieckenius de Souza*[1]
*Violeta S. T. D. B. Quental*[2] *Daniel Schwabe*[1]

[1]*Departamento de Informática*
[2]*Departamento de Letras*
*email: [ceciliak, clarisse, schwabe]@inf.puc-rio.br,*
*violetaq@let.puc-rio.br*

PUC-RioInf.MCC04/00 January, 2000

**Abstract**

This paper proposes representation dimensions of models of eXtensible Web-based Information-intensive Task-oriented Systems (X-WITS). The proposal is expressed in a unifying language that acts as the base for the construction of extensions and generation of texts, which are descriptions or narratives of interaction flows between end-users and X-WITS. The general architecture of X-WITS is presented, as well as a walkthrough of an extension scenario which enlightens the main characteristics of the proposal. The proposal is then discussed in view of its theoretical underpinnings: Semiotic Engineering and End-User Programming.

**Keywords**
End User Programming, Web based systems development and maintenance, Augmented Transition Networks (ATN) , Workflow Systems

**Resumo**

Este trabalho propõe dimensões de representação de modelos para sistemas extensíveis, baseados em ambiente WWW, intensivamente informacionais e orientados a tarefa (X-WITS). A proposta é expressa em uma linguagem unificadora que serve de base para a construção de extensões e a geração de textos descritivos e narrativos de fluxos de interação entre usuários e X-WITS. A arquitetura geral de X-WITS é apresentada, assim como um cenário de extensão que ressalta as principais características da proposta. A proposta é então discutida à luz de sua base teórica e de trabalhos relacionados da área de Engenharia Semiótica e Programação para Usuários Finais (*End-User Programming*).

**Palavras-chave**
Programação para Usuários Finais (EUP), Desenvolvimento e Manutenção de sistemas baseado na Web, ATN, Sistemas de Workflow

Sponsored by CNPq, TeCGraf and Petrobras

## 1. Introduction

This work is motivated by the fact that it is nearly impossible that a single piece of software will provide solutions for every specific problem a particular user may have in a given domain of activity [Myers]. Although this is widely known and accepted by software designers and developers, it is nevertheless true that extensible applications (i.e. applications that users can themselves reprogram and change) are difficult to design, develop, and use.

Our own work in Semiotic Engineering [de Souza'93, de Souza'97, Barbosa&de Souza'99, Barbosa&de Souza'2000] has explored some aspects of human meaning assignment processes that shed light on the nature of difficulties posed by extensible applications. They stem from the fact that human interpretation of signs (e.g. user interface signs appearing on computer screens) yields an indefinitely long chain of associations among other signs. For instance, in view of the left arrow sign BACK found in most browsers, users may interpret it as meaning GO BACK, which is associated to WHERE I WAS, which is associated to WHAT I WAS DOING, which is associated to WHAT I INTENDED TO ACHIEVE, and so on. In other words, any of the associated meanings assigned to the left arrow BACK sign is as good as any other, from a "subjective" point of view. However, from an "objective" point of view, the implemented meaning for the sign is determinate and unique: navigate to the previous URL mounted on this window space. This navigation may not correspond, for instance, to WHAT I INTENDED TO ACHIEVE.

Another example of this process in action (technically called "semiosis" [Peirce, Eco'84]) is when users take an editor such as MS Word (primarily designed for text editing purposes), and wield it for preparing slide presentations or for designing data capture forms. These related usages of the editor not only attest the wide spectrum of applicability [Fischer] of this piece of software, but they also give evidence of an indeterminate semiosic path evolving from text editing: an association with slide preparation and drawing, with form-filling, and other possible connections not mentioned here.

Thus, the applicability of software is the result of semiosis as performed by users, designers, or both communities. They all generate potential requests for maintenance or for new software releases (or products). The more popular an application, the greater the number of users engaged in semiosic processes, probing the limits of software applicability. This may generate unwielding loads of maintenance and/or redesign requests.

End-User Programming (EUP) environments can be regarded as an indirect form of negotiation about meanings between end-users and application designers. By developing extensions, end-users are metaphorically writing a message back to the designers about how they receive and evaluate the application, and about how it could be enhanced to meet some unpredicted need. EUP is thus part of the solution to taking some of the maintenance burden off the shoulders of software designers. EUP can also be a more satisfying alternative to end-users themselves, who won't have to wait long for some desired features.

With this scenario in mind, we are developing an EUP environment for Qualitas, a Web Based Workflow System currently running in its beta release for a community of about twenty users engaged in teamwork. Supporting EUP in this environment requires enriched representations for activities, tasks, roles, interaction and information structures.

It is important to note that, in this context, there is another fundamental aspect that has to be considered: the usability of such representations as perceived by end-users. We will separate two different dimensions of usability in this case. On the one side, we place cognitive issues related to how easy it is to understand what can be done, how to do it, and how to express what one wants to do. On the other side, we place pragmatic issues related to the range of operations an end user can perform and the mechanisms by which this is achieved. In this paper, we will emphasize the latter, and not discuss the former (which is the object of future work). More specifically, we will describe a representation language that integrates parts of the application's domain model (such as tasks and roles) and the interaction model (such as interfaces and documentation). This language is a kind of "lingua franca" for designers, developers and empowered end users (i.e. a language they can all use for their own purposes). From it, lower-level implementation code or schemata can be derived, as well as natural language texts that can explain and/or describe aspects of the application's model. It also plays the role of a common linguistic reference for translations to and from other interface languages such as the ones we should build for a complete and fully usable EUP environment.

The paper is structured in four sections, the first being this introduction. Section 2 presents a detailed extension scenario in Qualitas's domain. It highlights EUP dimensions accounted for by our representation language and provides a reference for the subsequent parts of the paper. Section 3 describes the general architecture proposed for eXtensible Web-based Information-intensive Task-oriented Systems (X-WITS) and presents the top-level specification of our representation language. Finally, Section 4 presents a discussion of our proposal in view of its theoretical underpinnings and related work on end-user programming.

## 2. An Extension Scenario in QUALITAS

The main purpose of Qualitas is to assist its users in following departmental policies that guarantee ISO quality standards. According to these policies, some of the employees' tasks are seen as services that they provide to customers. To guarantee the quality of each service, an employee, a technician, basically has to perform the following steps, after a customer requests a service:

1. To draft a service proposal
2. To present the proposal to the customer (a negotiation may be involved in this step, causing several versions of a proposal to be elaborated, but this has been abstracted away here in order to simplify the example)
3. To turn the proposal into a Contract, once the customer has accepted the proposal
4. To prepare for the execution of the service documented in the Contract
5. To execute the service
6. To document the Customer evaluation of the service execution.

Notice that all these steps are performed in Qualitas by the technician. This means that every time a customer has a request, he[1] has to get in touch with one of the technicians, who enters the request information into the system. Let us suppose now that there is a technician who is having trouble to get her job done because she receives too many phone calls. This person wants to allow customers to post their requests directly into the system. We will thus examine how the technician could extend Qualitas in order to reach this goal.

[1] For the sake of clarity, during this scenario we will be using the masculine gender to refer to the customer and the feminine gender to refer to the technician

The general idea is to allow a customer to enter some of the proposal information directly into the system. This could be done by copying and then adapting the system interfaces (web pages with HTML forms and/or messages) that compose the steps 1 and 2 listed before. In our proposal, the extensions are added to existing functionality; they do not replace it.

When entering the Extension Environment, the technician navigates looking for the desired interfaces. The navigation path reflects Qualitas' workflow representation (depicted in figure 10, in section 3). At first, she chooses one of the business processes to work with. The business process is the top-level representation of the work processes pre-defined in the workflow system. Figure 1 shows Qualitas' business processes. We will be focusing on the "Contract Service" process.

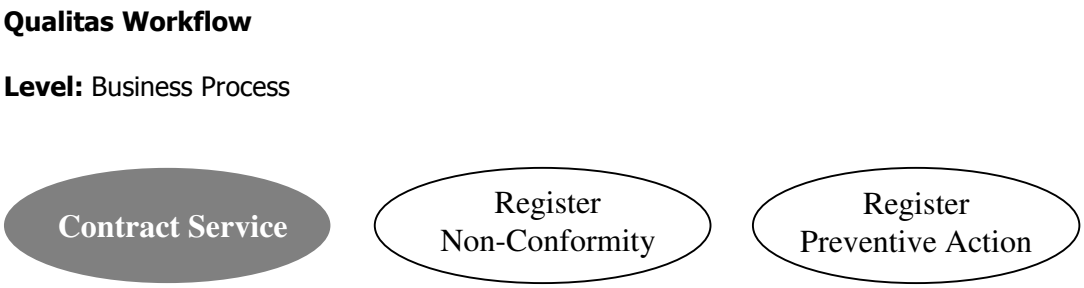



Figure 1. Qualitas' Business Processes

Business processes are composed of a series of coordinated activities. Each activity has an associated set of roles (e. g. "technicians", "administrative personnel") that can perform it. Instances of activities are assigned to people that hold a compatible role with the one specified for the activity. One person may play more than one role and two different people may play the same role.

Figure 2 shows the coordinated activities that compose the "Contract Service" business process. In workflow terms, this coordination of activities is called an Activity Network.

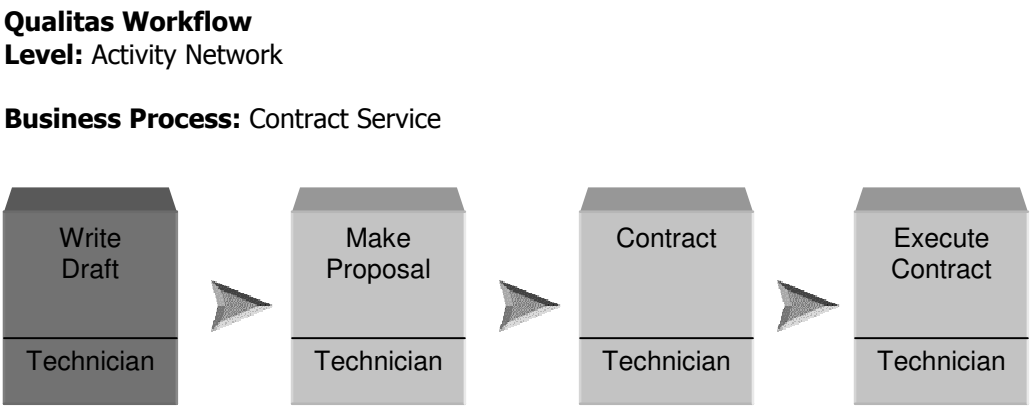



Figure 2. Contract Service Activity Network

The Activity Network can be much more complex then the sequence represented above, which has been simplified for our example. The representation shows which roles can perform each activity. For instance, the activity “Write Draft” can only be performed by Technicians, but could not be performed by Coordinators, among others.

Our user continues the navigation. She chooses the activity “Write Draft”, and reaches the Interface Level. The activity “Write Draft” is sub-divided into the Interfaces presented in Figure 3.

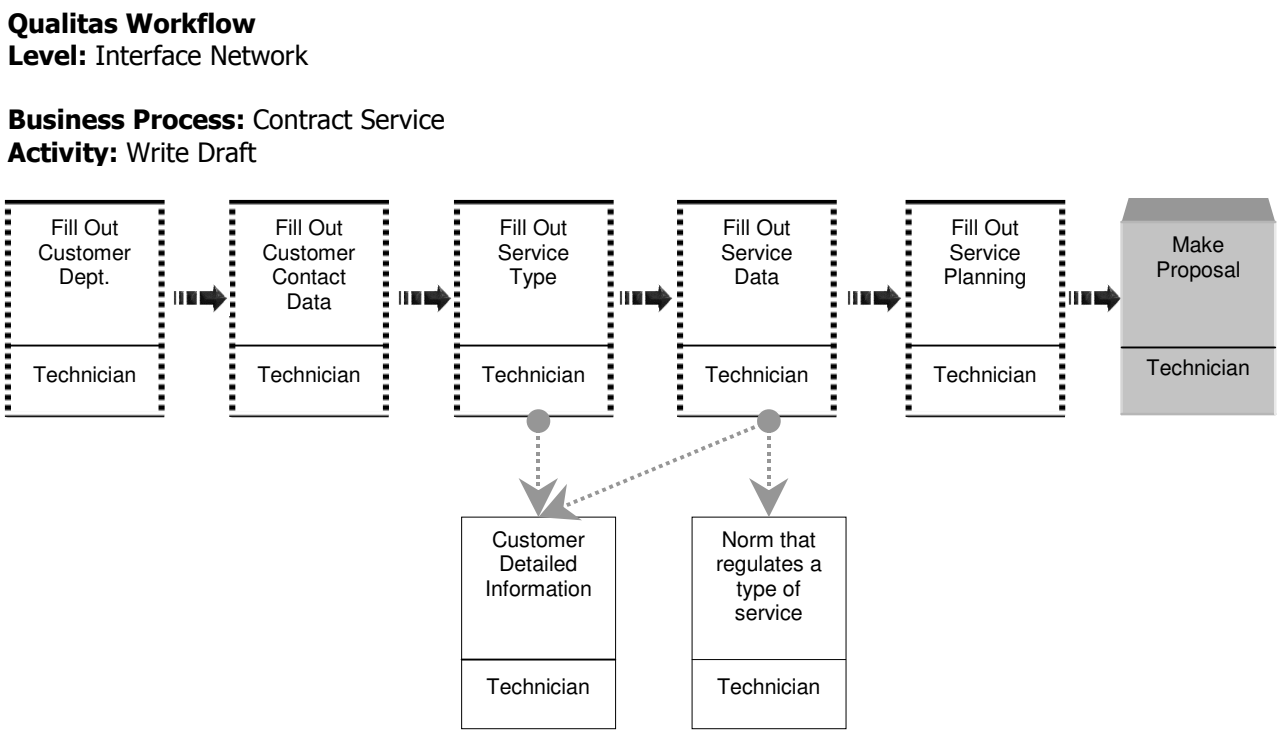



Figure 3. Write Draft Interface Network

Each box, except for the last one, represents an interface[2] (web page) that appears to end-users as they interact with the application. Each box is divided into two parts. In the upper part, the interface title is shown. This title is the same that appears to end-users when they are interacting with the application. If this is not enough for the user who is making the extension to remember what the interface looks like, she can click on the box and the interface layout is shown. In the lower part, inherited from the corresponding activity in the Activity Network, the user can see the roles of employees that can interact with the interface. The horizontal arrows indicate system processing of the previous interface. They will also be called actions. The vertical and diagonal arrows indicate hypertext navigation.

Note that the horizontal arrow after the last interface “Fill Out Service Planning” actually represents a sequence to the next Interface Network, which is the Interface Network that belongs to the activity “Make Proposal” (the successor of "Write Draft" in the activity network shown in Figure 2).

[2] As pointed out before, the interfaces we are currently using are HTML forms and hypertext web pages. Multimedia and hypermedia interfaces are not included yet.

Extensions take place at the Interface Level. In other words, we propose an interface-driven approach to extend this kind of application. The user constructs extensions by building new Interface Networks, and consequently new workflows. This can be achieved by selecting and copying the desired interfaces and arrows in a composition process. As will be shown, during extension, the user may adapt the copied interfaces or actions that process these interfaces.

In this scenario, our user would like to copy the sequence of the first three interfaces: "Fill Out Customer Dept.", "Fill Out Customer Contact Data" and "Fill Out Service Type" and their corresponding actions. Figure 4 shows a schematic view after the user has done this.

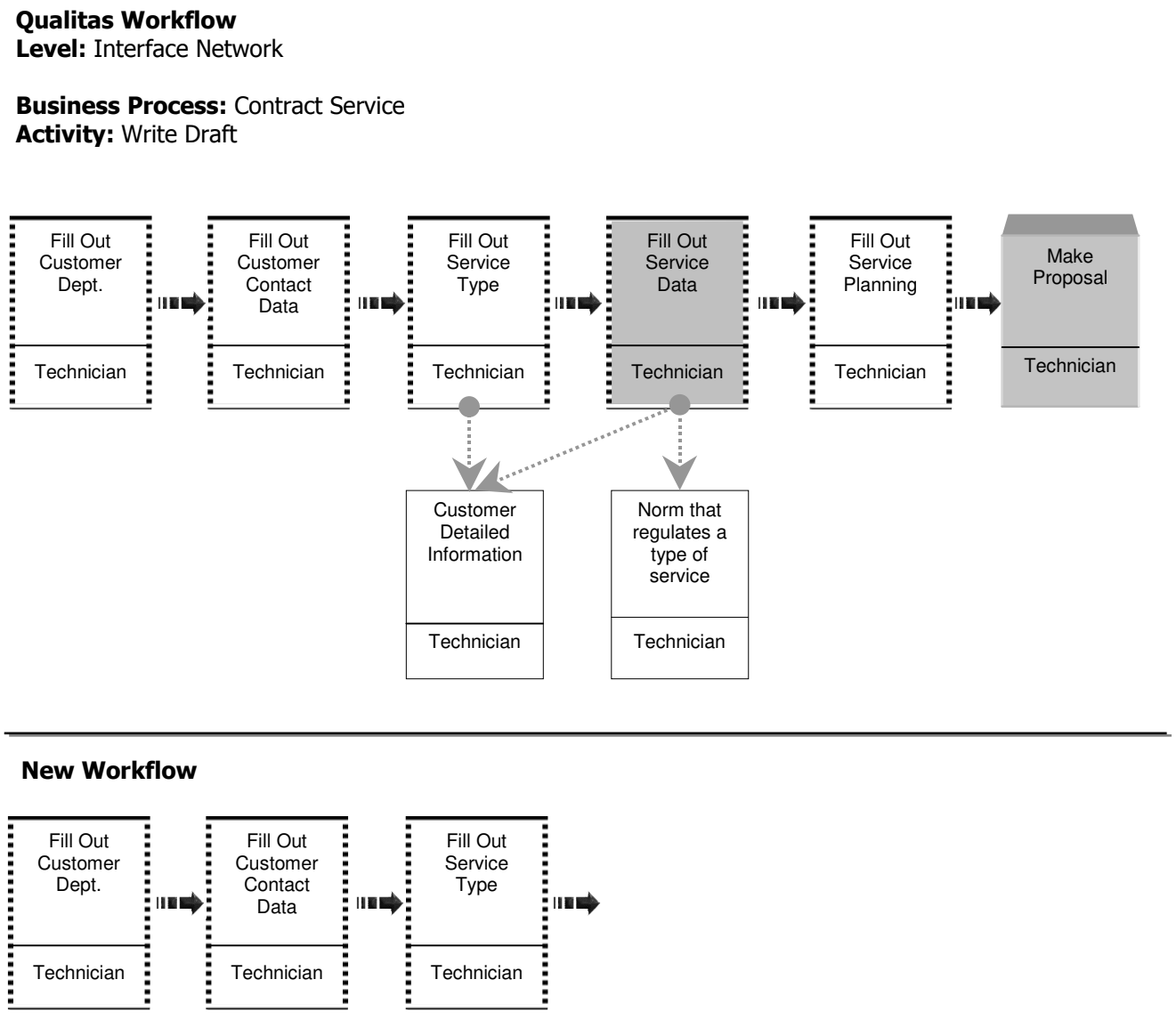



Figure 4. Creating the New Workflow

Note that the user did not copy the interface called "Customer Detailed Information" and the vertical arrow that connected it to the interface "Fill Out Service Type". In this new workflow, the customer should be directly entering service information. Thus, our user decides that the customer

does not need detailed information about himself. This is a possible way to "turn off" or eliminate existing navigational links. Later on we will show an alternative way to do this. It is important to mention that the user is also able to "turn on" or create navigational anchors to existing system information.

At this point our user has managed to create a new workflow. She can already imagine a customer informing, at first, his department, then informing contact data (such as name, phone number and e-mail) and then deciding what type of service is being requested (education, consulting, etc.). But the user also notices that only Technicians can interact with these interfaces (lower part of each box). So, she decides to change this in the new workflow in order to allow customers to access the interfaces. She chooses one of the new interfaces and then accesses the interface property **Role**. But the problem is that the allowed values for this property are only: "Technicians", "Coordinators", "Head of Department" and "Administrative Personnel". "Customer" cannot be indicated as a **Role** that can interact with an interface.

At this stage our user has reached a point where the extension task has to be shared with the application developer. It is important to mention that the idea of our proposal is to induce end-users to extend the application and keep consistency with the original design. In semiotic terms, users are guided to preserve consistency with the original designer's message. In order to maintain this consistency, users should be able to adapt the message, but not to replace it or introduce negations of it. Although there is no guarantee that users will always be consistent with the original message, our solution guides users in order to do so. As a consequence, some restrictions are imposed on the set of possible extensions. These restrictions are that end-users are not capable of creating, from scratch, structurally new objects, relationships between objects or behavior. As mentioned, extensions are always based on existing and functional solutions. Extensions are an adaptation or a different combination of existing solutions. Users may, as will be shown during the continuation of this example, create new objects that are structurally equal or similar to existing ones, or change the processing that takes place between two interfaces, always taking the existing representation as a basis.

The creation of the new role "Customer" implies the creation of new relationships, attributes and behavior for this object. A new relationship between the "Customer" object and the "Contract" object should be created. New attributes for "Customer", such as a password attribute, should be added as well as a new operation that supports the input of contract data by the customer. In semiotic terms, there would be a change in the meaning of object "Customer". It would be transformed from a passive object, about which information is stored, into an active object that can change information about other objects. This change of meaning is actually a distortion of the original designer's message and thus is not allowed in the proposed extension environment. As future work we plan to evaluate how the metaphoric mechanisms proposed by [Barbosa, Barbosa&deSouza'99, Barbosa&deSouza'2000] could be used in order to provide a solution to this case, where the customer would probably be regarded as a "technician that does not execute services, nor belongs to same company hierarchy and department".

Consequently, our user gets in touch with Qualitas' developers and they decide that the user should continue extending Qualitas, using Technician as a temporary **Role**. In the meantime, designers will create the new role "customer" and build the mechanisms that guarantee customer access to the system.

Continuing the extension, our user's focus now is on the interface "Fill Out Service Data" (see Figure 4). This interface captures details about a service. From the user's point of view, the

problem is that some of these details should not be informed by the customer. For example, in an education service, the technician informs whether slides will be used, what kind of material will be provided to the students, etc.

This interface also has navigational anchors to the "Customer Detailed Information" and to the "Norm that regulates a type of service" web pages,represented by underlined contents, which should not be made available to the customer in the new scenario. Thus, our user would like to adapt this interface in order to use it in the new workflow. To do this, the user acts upon the interface and interface fields. Figure 5 demonstrates part of the "Fill Out Service Data" interface. The navigational anchor to "Customer Detailed Information" is represented by the underlined content $|current_customer(name)|$. The part of the interface depicted in figure 5 does not include the anchor to the corresponding Norm. Note that each selected (highlighted) field has, at its right side, a set of possible changes that can be done by the extender.

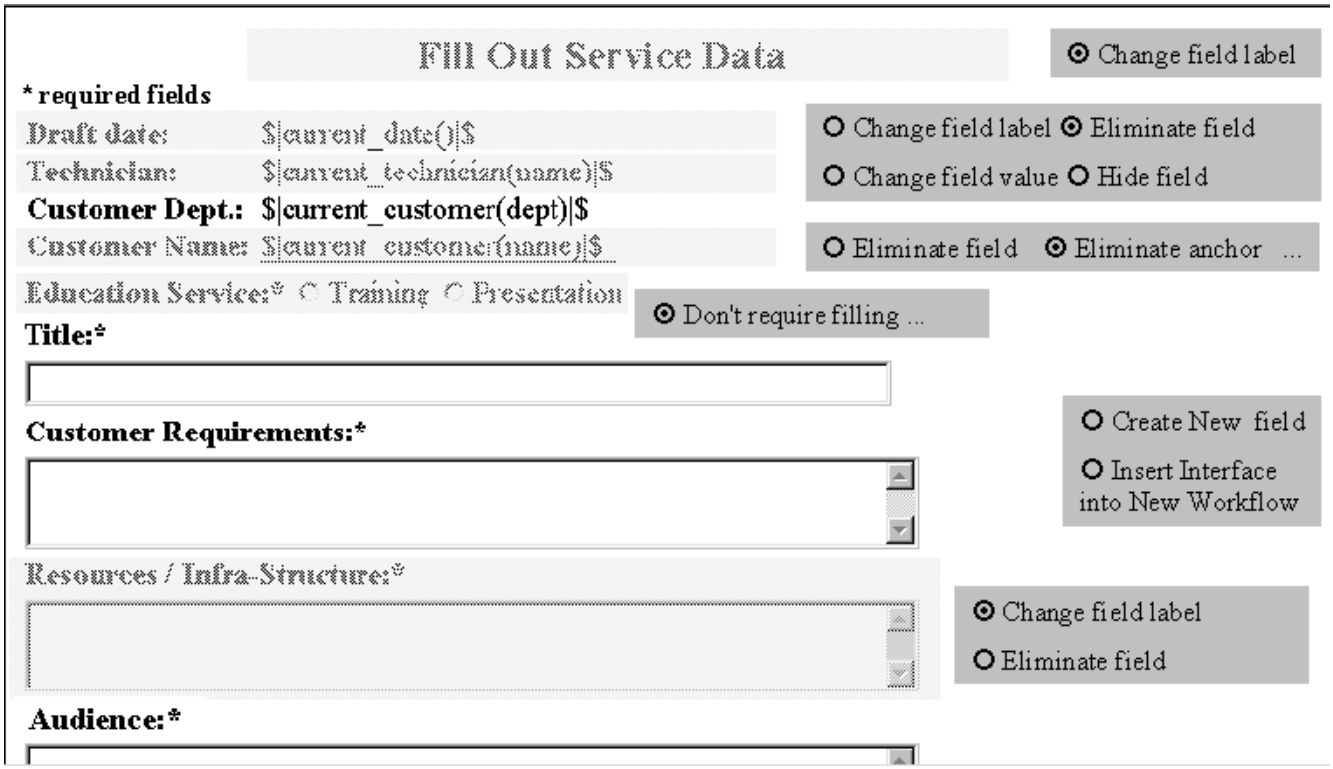



Figure 5: Adapting an interface

After adapting this part of the interface, it would look like Figure 6. Note that the interface title was changed. The navigational anchor to Customer does not show up - the $|current_customer(name)|$ output field is not underlined anymore. The fields "Technician" and "Resources / Infrastructure" were eliminated. The input of the field "Education Service" is not required anymore, since the symbol * (that means required field) does not show up in the adapted interface.

Field elimination has an important consequence: it means that the eliminated information will not be available for the actions that will process this interface later on. The option "Hide Field"

allows an information to be available for future processing, without displaying its corresponding widget on the screen.

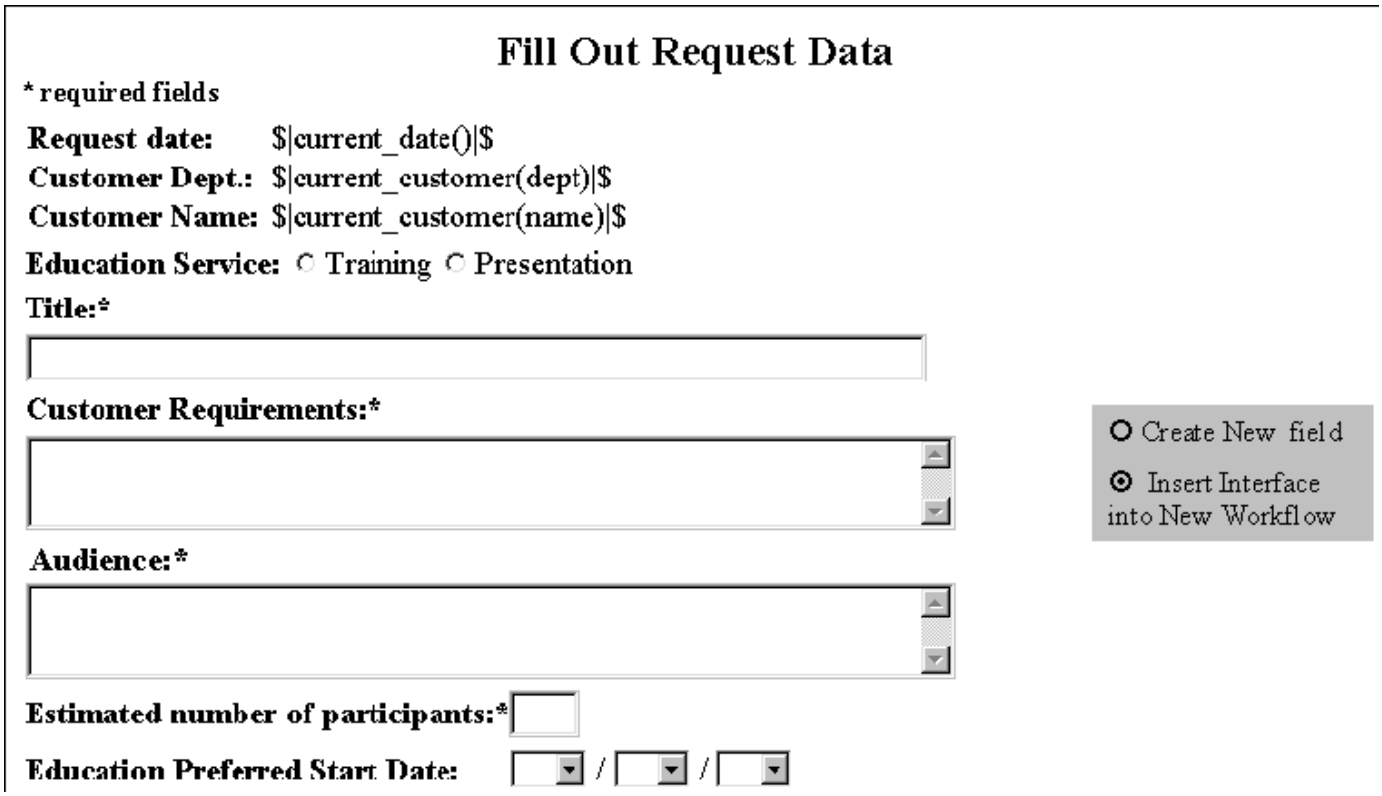


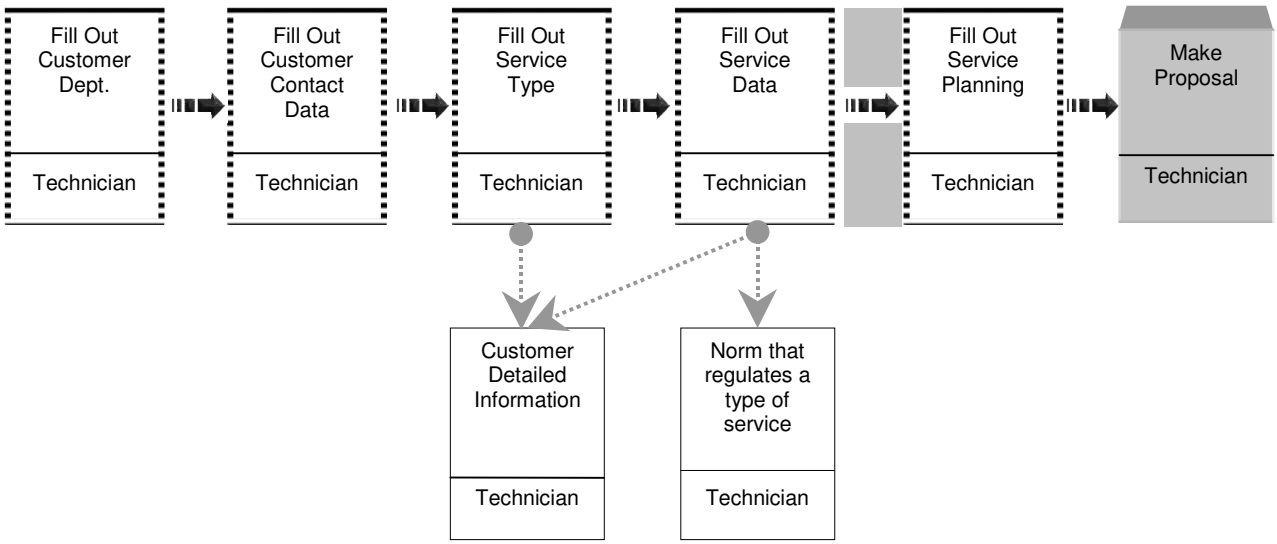

Figure 6: Adapted interface

By changing the interface title, the label that appears in the (last) box of the new workflow is also changed from “Fill Out Service Data” to “Fill Out Request Data” (see Figure 7). Semantically what happened was the creation of a new object, called “Request”, that is similar to the Draft object. Also note that in figure 7 the navigational arrows do not appear. They were eliminated during the adaptation of this interface - a different approach from the first one presented, in which the user simply did not copy the navigational arrows from the Interface Network.

Our user will now copy the action that processes this interface, highlighted in Figure 7.

**Qualitas Workflow**
**Level:** Interface Network

**Business Process:** Contract Service
**Activity:** Write Draft




**New Workflow**

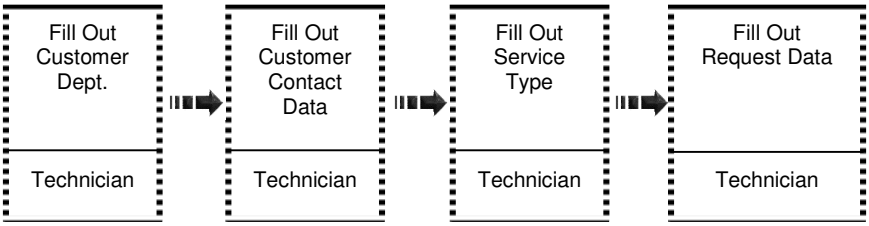



Figure 7: Copying an action

However, this action cannot be copied as is into the new workflow, mainly because the information of the eliminated fields is not available anymore. Each step of the action that referred to the eliminated information has to be revised. The extension environment guides the user during this revision, prompting the user about the problems encountered and proposing solutions such as the following:

*This action may only be executed if the following rule is satisfied: "There is a technician responsible for the service."*

*The information "technician" is not available for this action. The possible solutions are:*
1. *Make this information available by adding it to the previous interface*
2. *Make this information available by assigning a fixed value to it*
3. *Do not check the rule*

*Information expected to be saved into the database is not available. Please inform how to proceed in each case.*

***Resources / Infrastructure:***

1. *Don't save this information into the database*
2. *Make this information available by assigning a fixed value to it*
3. *Make this information available by adding it to the previous interface*

...

Instead of trying to copy the action directly, the user could also have chosen to adapt it first, which would have avoided these messages.

Continuing the extension, the user wants to skip the interface "Fill Out Service Planning", since it is not of interest to the customers. Now the user wants to turn the Request (the Draft-like object) into a Proposal. This would be a way to connect the new workflow with Qualitas' original workflow. The request would become a proposal, which would later be transformed into a contract, which would be prepared, executed and evaluated. This change of status would also guarantee that the customer would receive an answer about the request made in five working days (this is the basic difference between a Draft and a Proposal).

The user goes back to the action just adapted and tries to add a step to it that turns the request into a proposal. But the system warns that in order to become a proposal, the request should have a responsible technician. Thus the user decides to create a new interface where a technician would be assigned to a request.

The basic method for creating a new interface is to compose it by choosing interface widgets and then associating each widget with system information. Figure 8 shows a possibility for achieving this. On the left side, we see the interface under construction with its corresponding fields. On the right side, in a gray background, the user associates these fields with system information. The user also has to inform the roles that will be allowed to interact with this interface.

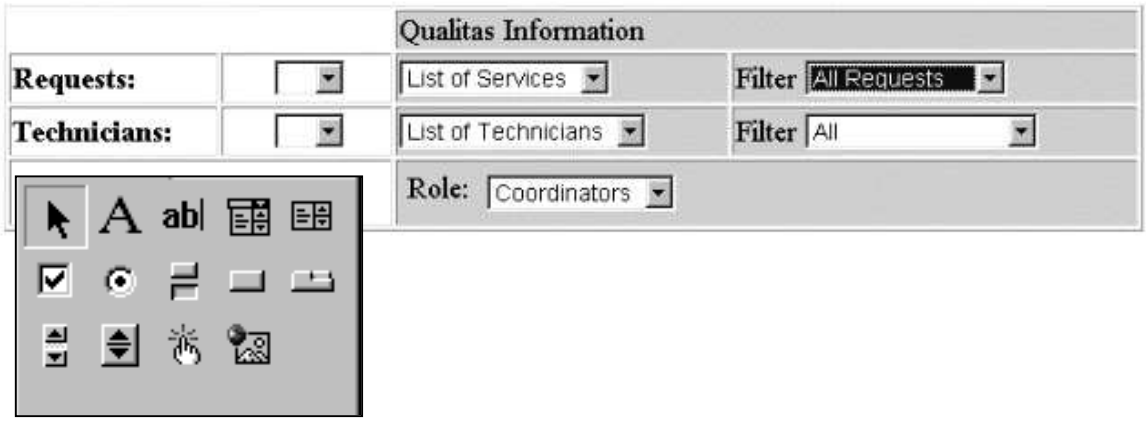



Figure 8. Creating a new interface

At this point our user decides that only Coordinators would have the right to allocate technicians to customers' requests. It is important to point out that in this extension environment users can make extensions whose parts are targeted at users with roles different from their own. In order to

allow this, the extension environment has CSCW (Computer Supported Cooperative Work) mechanisms that guide the user during the extension, so that users manage to do it in a collaborative way. These mechanisms are outside the scope of this paper. It is a study in its own that will be described in future work.

After composing the new interface, the user creates the action that will process it. The user is provided with a set of steps and builds the action by choosing some of them. She also informs how to process each information available so far (usually originated at the interface). Figure 9 gives an idea of how this could be possible.

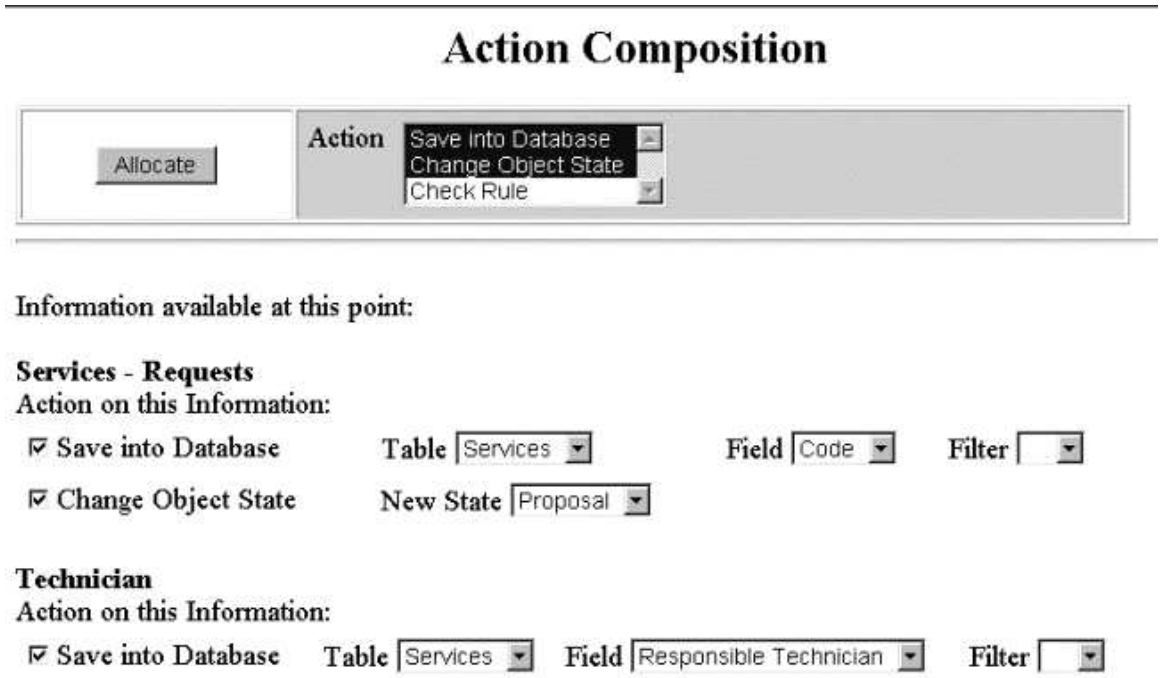



Figure 9. Composing an Action

The user-composed action basically saves the information into the database and transforms the request into a proposal.

After doing this, the user considers the extension to be finished and ready to be tested. Tests are handled in a test environment, separated from Qualitas itself, in order to protect the original system. When the user is done testing, she can make the extension available at the application interface.

During execution, problems not anticipated by users who made the extension might happen, such as a power failure or exceptions occuring during the execution of an action. Exception Handling is an important and complex subject which deserves special attention. In future work we plan to discuss how the X-WITS proposal deals with exception handling and its different facets [Chiu&Karlapalem&Li].

## 3. X-WITS Architecture and Representation Language

This section describes the general architecture proposed for eXtensible Web-based Information-intensive Task-oriented Systems (X-WITS) and presents the top-level specification of our representation language. The general architecture is composed of two models:

- The Domain Model, which is sub-divided into:
  - a Task Definition Representation, and
  - an Information Representation; and
- The Interface Model.

The main focus of this paper is on the representation language we propose. This language integrates the elements of the different models and ultimately specifies the interpreter of our extension environment. It is actually the extension language per se, which will be used for more than one purpose: extensions and explanations or descriptions about the system's underlying model and functioning.

### 3.1 The Domain Model

#### 3.1.1 The Task Definition Representation

The task definition representation is a representation of the tasks inherent to the domain. Currently, our model is applied to workflow systems, which require not only the representation of tasks, but the roles that perform these tasks. So we will concentrate on the description of the representation of a workflow domain. As future work we will evaluate the application of our model to non-workflow / non-groupware domains, which we believe will simplify the representation imposed by a workflow domain. Our current intuition is that the proposed model is not restricted to workflow applications.

A Workflow is an automation of business procedures “where documents, information or tasks are passed between participants according to a defined set of rules to achieve, or contribute to, an overall business goal” [WMC’1995].

The representation of Workflow used here is a top-down representation, such as in [WMC’1999], where, at the top level, there is a set of business processes (see Figure 1). At the next lower level, each business process is represented as a network of activities (as in Figure 2). An activity represents a logical step within a process and has a defined set of roles responsible for performing it. The role of responsible for a task or activity can be assigned to the participants of the workflow (or users of the workflow system). By using this representation, the actual participants of an activity may change without affecting the workflow representation and processing.

In order to support the extension environment, one more level of detail is added to this representation. Each activity is represented as a network of interfaces. Each interface is a representation of the actual system web page (screen). The interface inherits the role from the corresponding activity (as shown in Figure 3). When interacting with an interface, end-users may trigger system processing, called “Action” in our representation, which computes the data originated from the current interaction in combination with possible available information, derived from previous interactions, pre-requisite of the current one.

A generic representation of a workflow domain is depicted in Figure 10.

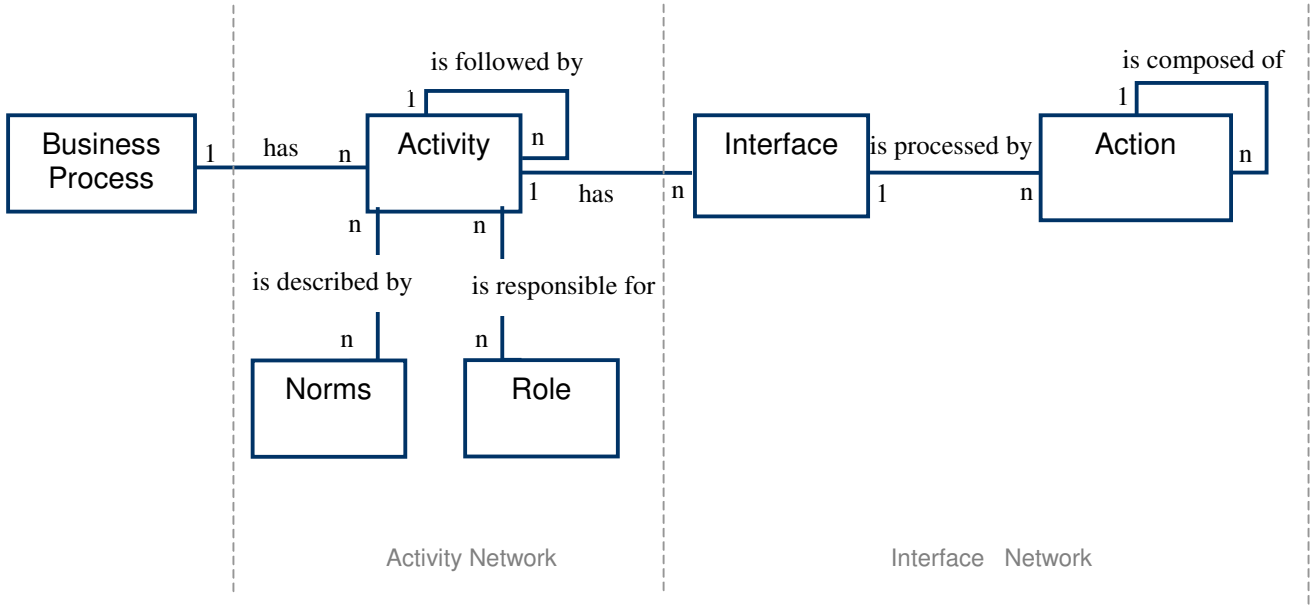



Figure 10. Workflow Domain Representation

The Interface Network is processed by the representation language that will be described in section 3.4.

#### 3.1.2 The Information Representation

The domain information is represented as information units that are related to each other. Each information unit has different perspectives that are used depending on system processing needs in a given moment. For example, when generating a natural language (NL) text for summaries or system messages, information like noun's gender is needed. When saving information into a database, information such as table and column name is needed. When adapting an interaction during extension, the interface perspective is used. Figure 11 shows how the Customer information unit would be represented.

The natural language generation (NLG) and the navigational perspectives play an important role, since they give support to the information intensive facet of X-WITS. When an end-user is involved in a task, interacting with the system, information related to the specific task is made available upon request, especially for decision making. The two perspectives mentioned allow detailed domain, semantically related information to be obtained from the current interface. For example, each interface may provide a link to the norm that regulates the activity a technician is performing. Or, if the process deals with the customer object, an anchor to information about the customer should be available (e. g. a technician might want to check information about the customer who is requesting a service, during the draft or contract phase) and this information can be provided by an automatically generated NL text.

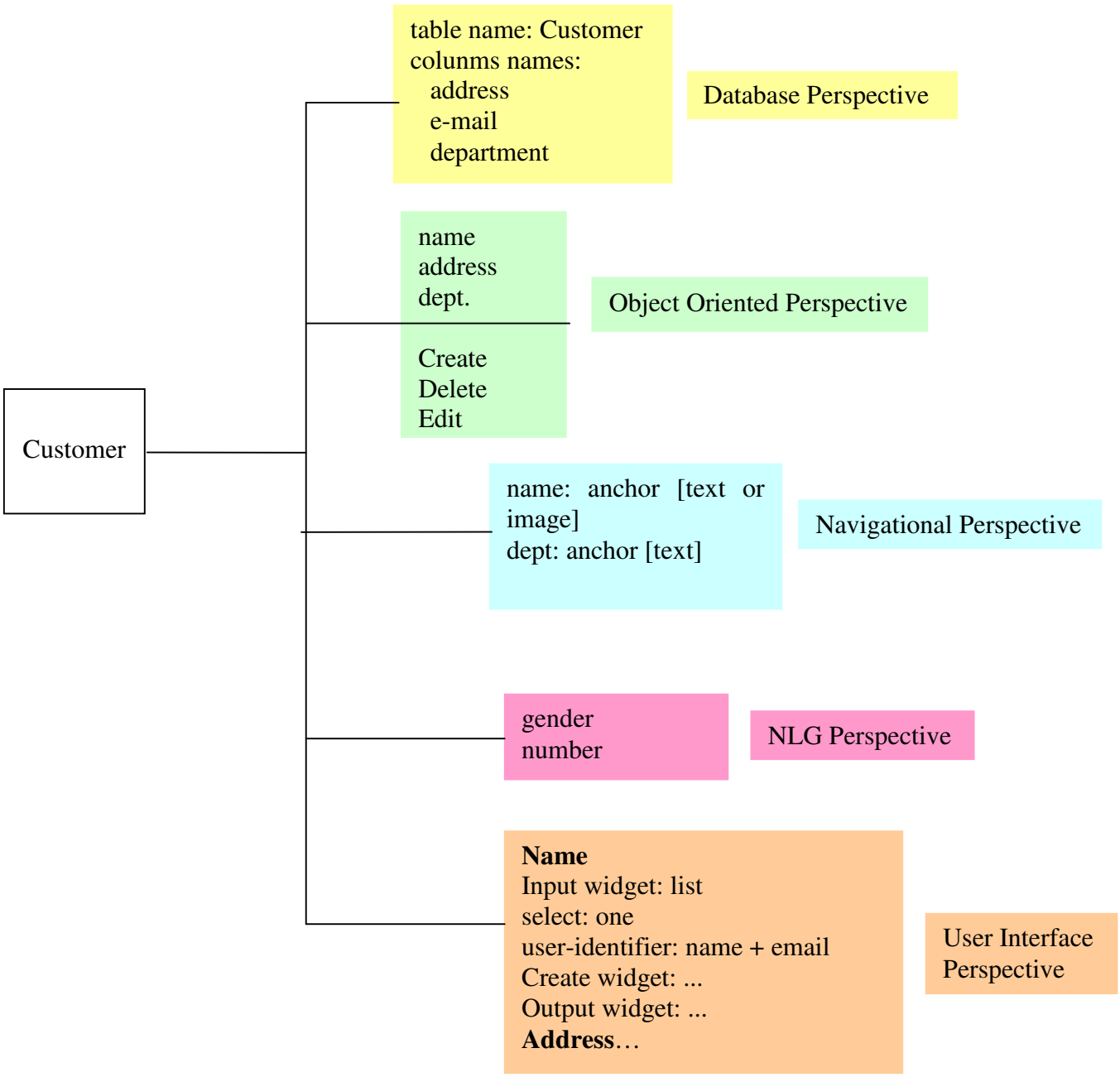



Figure 11. Customer Information Unit

Figure 12 depicts the main aspects of the Information Representation. Note that information units can be related to each other.

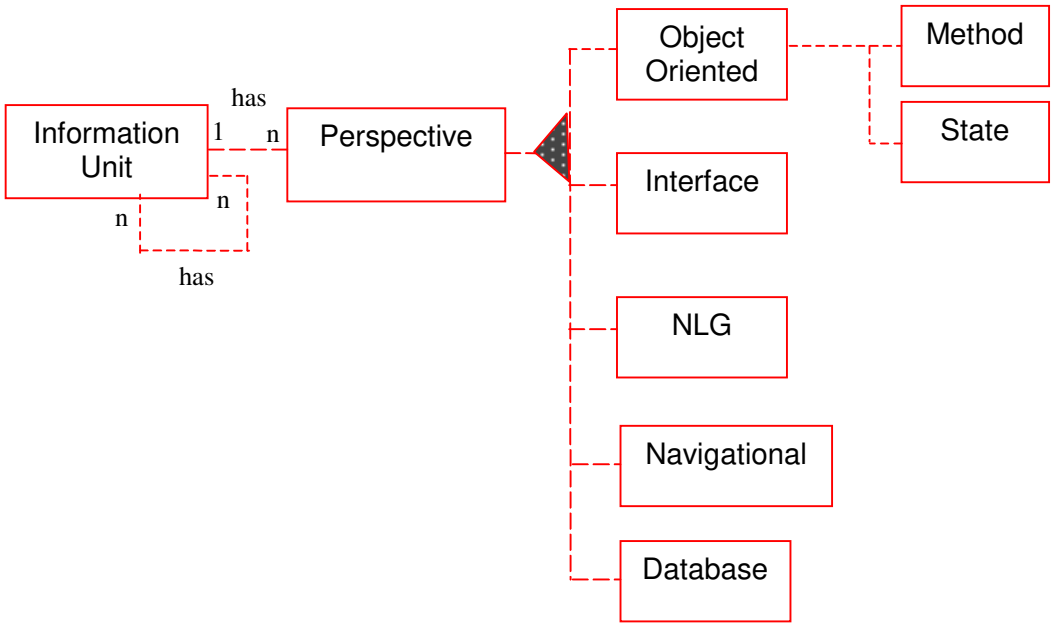



Figure 12. Concepts of the Information Representation

### 3.2 The Interface Model

The main purpose of the interface model is to enable end-users to adapt existing interfaces or to create new ones during extension. The full description of this model is out of the scope of this paper and will be detailed in future work [Dahis & de Souza, forthcoming].

Briefly, an interface is defined as a composition of fields. A field can be of one of three types: input, output or trigger. Input fields are data entry. Output fields are basically text that may represent:

- a field label, such as the text “Requests:” on the left side of Figure 8.
- a field value that may not be changed, such as context information originated by previous interactions. An example is the text $|current_customer(name)|$ in Figure 6.
- an automatically generated natural language (NL) text, such as error messages. Our model also allows the generation of workflow summaries and explanations. These summaries can be generated based on the internal representation language that will be presented in the next section. The general process is the translation of this representation language into a pseudo-NL text. The generation process is template-based. Advances in the NL generation process are future work, which includes the automatic generation of navigational anchors embedded in the texts (e. g. links), which are actually trigger fields.

A trigger field is a field that triggers the processing of the current interface and/or the display of the next interface, such as a button or a navigational anchor. Each field has a corresponding set of widgets that can be selected by the user during an extension. Figure 9 depicts part of the Interface Model.

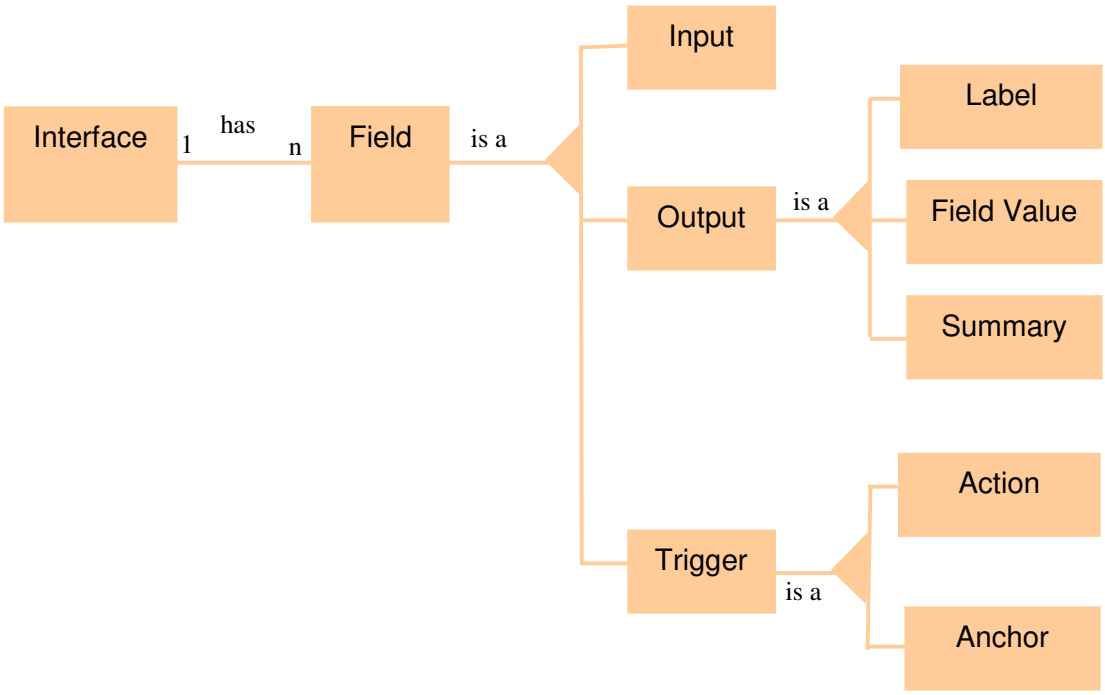



Figure 13. Interface Model

In an interface instance, each field is associated with a widget and a system information that is represented in the Domain Model. For example, navigational anchors always represent semantic relationships of the Domain Model, such as the Activity and Norms relationship depicted in figure 10, or a relationship between a contract and the corresponding customer or responsible technician.

The association of the Interface Model with the Domain Model is what enables the presentation of the extensions possibilities for each field of an interface, and the interface as a whole, as shown in figures 5, 8 and 9.

We will now outline how the Domain and the Interface Model relate to each other.

### 3.3 Relating the Two Models

The Information Representation serves both the Task Representation and the Interface Model, as depicted by in Figure 11.

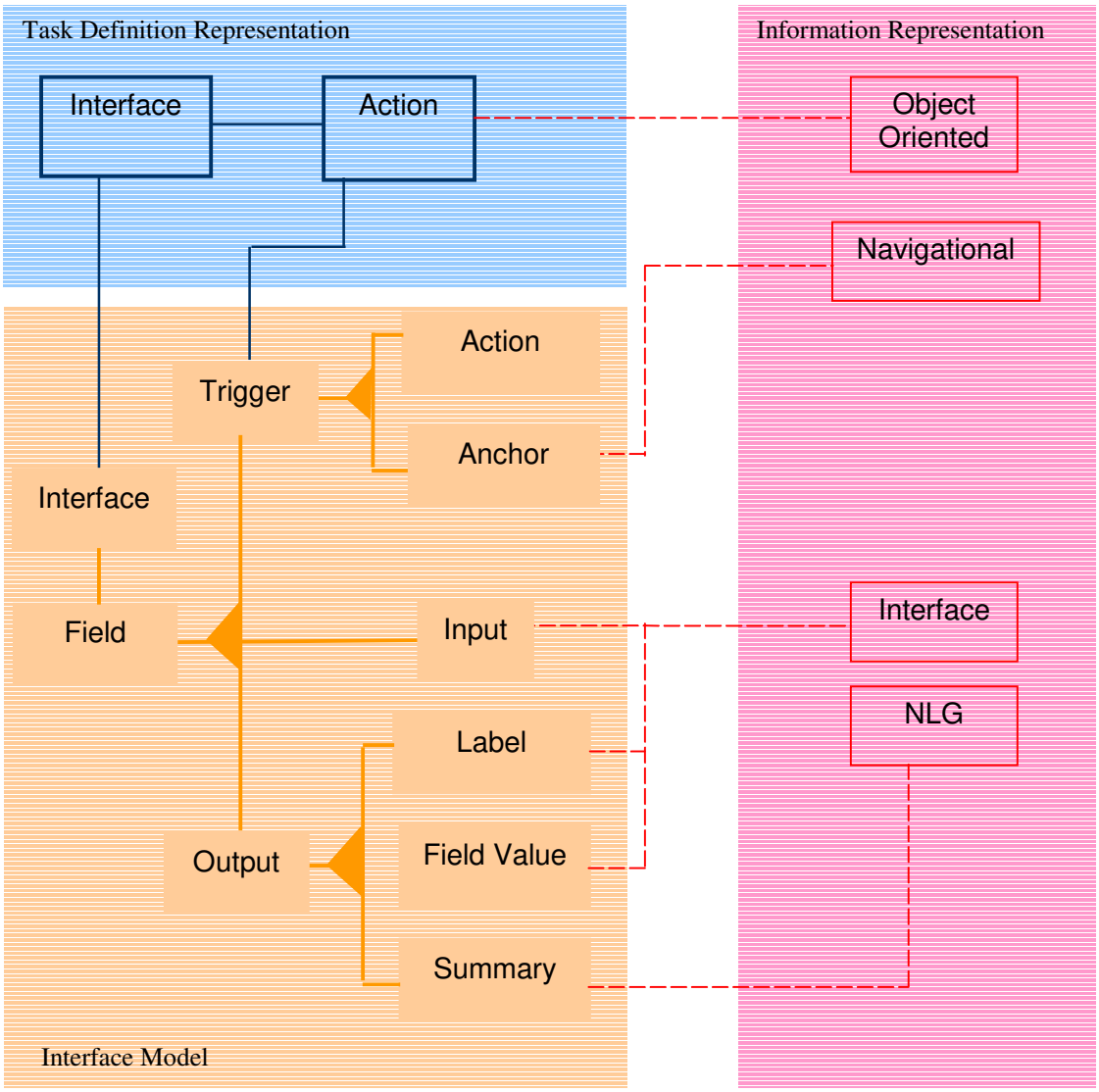



Figure 14. Relations among models

The methods of the Object Oriented (OO) perspective of an information unit compose the actions of the Task Definition Representation, which in turn are the actions available to the user when creating or adapting an interface. The interface perspective of the information unit helps to guide the user when mapping widgets (input and some output) to information units. The NLG perspective provides information necessary to text generation. The interfaces of the Interface Model are actually those that will be used in the Task Definition Representation. In addition to thesestructural relationships, all these models are integrated in the representation language we propose.

### 3.4 The Representation Language

The representation language is formalized as an Augmented Transition Network (ATN) [Woods], which stands for the processor of the Interface Network. The ATN is the extension language per se and will be used for more than one purpose: as the interpreter of our extension environment and as the source for explanations or descriptions about the system's model and functioning.

The use of grammars as formalisms to represent and process flexible workflows has been previously presented [Glance et al]. However, we propose the explicit representation of how the workflow is reflected at the interface level, thus enabling the interface-driven approach to application extensions. In Semiotic Engineering terms, the ATN represents not only the designer's message about the task oriented domain (the workflow), but also the connection between this message and the decisions about how the message should be conveyed to the end-users (the mappings with the interface). The major gain in this approach is that the designer's message about the domain and about how the domain should be expressed are connected and explicit. As a consequence, this enables end-users to manipulate the interface they know and, by doing that, provoke changes at the domain level – e.g. how the workflow is structured. Ultimately, this enables the interface-driven approach to EUP, which allows users to build on the knowledge they have about the user-interface to construct extensions, thus maximizing the usability of the extension environment.

One may argue that this direct and explicit mapping of the domain and the interface may compromise modularity and independence between the application and the interface. To deal with this, a more abstract and system independent interface representation could be used, and later on it would be mapped into the interface intricacies of the current system environment. Since our focus at the moment is on web-based systems, the specification of a system-independent representation of the interface at the ATN is part of future agenda.

Let us now explain the proposed ATN itself. Each state in the ATN represents an abstraction of an interface in the Interface Network. The various arc types are the actual representations of the structures, actions and sequences of interaction that put the interface network into effect. When moving from one state to the next, actions are executed and the actual interactions take place. These actions can be Transformational, Non-transformational or a combination of both. Registers in the ATN play a fundamental role in the process. They serve as (global) variables that will hold parameters shared by different states or interfaces. They also serve as a representation of information about domain and interaction. So, registers are themselves instances of parts of the Domain and Interface models.

The transformational actions of the ATN are:

1. Create a New Object (such as a Draft Proposal)
2. Destroy an Object
3. Substitute an Object by another (which is actually a combination of the previous actions). This action allows, for example, the substitution of a Proposal by a Contract
4. Multiply. This action is useful if there are several Proposals for a single service and one of them is chosen and contracted. Then all the versions that were turned down should be destroyed. This action is actually a combination of action 3, plus a sequence of actions 2.

The non-transformational actions are:

5. Store in Database
6. Retrieve from Database
7. Update Registers
8. Test. The set of tests we use are:
   8.1. Security (access control)
   8.2. Input validation
   8.3. Date limits (that have to be followed by the users in order to fulfill the organization policy)
   8.4. Pre-conditions for Destination State (each state can have pre-conditions that are required to be satisfied so that a transition to it can be completed. For example, the availability of information or input parameters)
   8.5. Synchronize
   8.6. Communicate
   8.7. Monitor

We would like to emphasize that some of these tests are executed against domain documents, their states and available information. These tests guarantee that some interactions are only possible or active if a document is in a certain state and/or has certain information in it. In Qualitas these documents are such as "Draft Proposals", "Proposals" and "Contracts". So, for example, the interaction and corresponding transformational action that turn a "Draft Proposal" into a "Proposal" can only be performed if the current document is a "Draft Proposal" and information like the Customer and the Service Type have been specified. This way, the representation language integrates the activity and document-oriented approaches to workflow [Abbott&Sarin]: by interacting with the application, a user fills-in documents and changes their states and, as consequence, certain tasks are made active or inactive. Differently from [Glance et al], we do not propose a dual representation of documents and tasks, but their integration through interaction steps. We can say that documents are evidences (or indexes, in semiotic terms) of past processes.

It is important to note that in the current domain the tasks map one-to-one onto independent sections of a document, characterizing the decomposable documents mentioned in [Glance et al] or the strict / fuzzy patch collaboration models in [Prates]. In future work we plan to test our proposal against non-decomposable processes [Glance et al] or the overlapping collaboration model of [Prates], where there is a many-to-one mapping of documents to tasks and tasks to documents.

Test actions 8.5, 8.6 and 8.7 handle parallelism. A more general framework and in depth description of this issue is also part of this research's future agenda.

The top-level rules contained in the ATN are of the sort listed below (in rewriting rule format, for sake of brevity):

task_interface_atn → arcset arcset*
arcset → #state# arc

```
arc      → 'cat' category test action terminate |
           'push' state test action terminate |
           'tst' label test action terminate |
           'pop' form test |
           'parallel' arcset arcset arcset*

category → 'data_entry' | 'output_message' | 'object_visualization' | 'menu'

test → 'security' | 'validate_input' | 'synchronize' | 'communicate' | 'monitor'

action →   'setr' register form |
           'sendr' register  form |
           'liftr' register form |
           'setdb' dbitem form |
           action*

register → #tmpvariable# | #information unit#

form → 'getr' register |
       'getdb' dbitem |
       #form_input# |
       'buildq' #structure_name# register* |
       form* |
       'quote' executable_code

dbitem → 'table' #table# 'field' #field# ['where_clause' #parameters#]

executable_code →  'destroy' #object#  |
                   'substitute' #object#  |
                   'multiply'  #object#  |
                   'generate_htmlform' #parameters# |
                   'generate_htmltext' #parameters# |
                   'visualize' #parameters# |
                   'generate_nltext' #parameters# |
                   'generate_navigational_anchor' #parameters#


terminate → 'to' #state#  post_test

post_test →  #pre_cond_for_nextstate#
```

where:

| represents alternative ways of forming a construction
* indicates zero or more repetitions

The ATN node types and formants used in our extension language specifications are:

CAT is an arc that can be followed if the current input symbol is a member of <category name> and the <test> is satisfied. For this arc, the actions are structure-building actions and the terminal action specifies the state to which control is passed as a result of the transition. Only CAT arcs generate workflow-related HTML interface pages.

PUSH and POP arcs transfer between networks.

TST is an arc that performs internal application tests, and only generate message interface pages in case the test fails. In other words, these are error-handling arcs.
SETR sets the value of the register at the current level equal to the value of <form> register information representation that is built as the ATN proceeds from state to state
SENDR sets the value of the register at the next lower level equal to the value of <form>
LIFTR sets the value of the register at the next higher level equal to the value of <form>
SETBD saves the value of <form> in the specified <table, field> of a database
GETBD reads the value in the specified <table, field> and make it available as <form>
TO indicates that the input pointer is to be advanced, that the next state is to scan the next input data. This will happen if the test in <test_ pre_cond_state> is satisfied
form as the <action> symbol, it represents a function call, where the first element represents the name of a function and the remaining ones, its parameters
test conditions, also represented as function calls
GETR is a function whose value is the contents of the indicated register
BUILDQ is a structure building form. It takes a structure with specially marked fields and returns the same structured with the marked fields filled out with the contents of indicated registers. Note that these structures are created in accordance with the representation specified in the Information Representation (e. g. an instance of class Customer, with some attributes filled out). So BUILDQ can create new objects of the Domain Model
QUOTE produces as value the argument form

Items initiated by "#" in the rewriting rules designate productions that will not be specified further in this ATN, either because they refer to terminal symbols composed of arbitrary alphanumeric strings or because they require specification involving structures that are too complex and outside the focus of our current discussion.

In order to illustrate how the ATN is put to work in extension tasks, we will show how it describes the first part of the Interface Network in Figure 3. We will start by showing how each state in the ATN generated description is an abstraction for the interaction to follow (which is specified in the arc sets exiting that state). We will also show how the ATN integrates the Domain and the Interface Model.

```
"Fill in Customer Department"
cat(Data_Entry,
     test(security),
     (
     action(setr 'Interface'  getdb [table: 'Customer' ;field:'CustDept']),
     action(setr 'webpage01'  quote generate-htmlform ['Interface']),
     action(setr 'CustDept' form_input),
     action(setr[Customer],buildq[CUSTOMER_OO.Dept,getr[CustDept]]),
     )
     terminate(to "Fill in Customer Contact", post_test)
    )
```

"Fill in Customer Department" is the name of our initial state. It has only one CAT arc associated to it. TST arcs for error-handling are omitted for sake of clarity. This arc is traversed if a security test is performed successfully. It tests whether the role that the user is playing is compatible with the interface network s/he is starting to follow. Information from the Task Definition Representation (roles that can interact with an interface) and from the Information Representation (role that the current user holds) are used at this point.

Traversal triggers a set of 4 actions: getting information from the database about the interaction patterns associated to Customer Department; generating an HTML form based on the retrieved interface database item and assigning it to an indexed webpage (under construction) - these mechanisms are provided by the Interface Model; reading input from the form and assigning it to a temporary variable; and finally building a new Customer information unit and assigning the register variable to the department attribute of its OO perspective. This actually represents the creation of an instance of Customer in the Domain Model. Termination of traversal is achieved with an indication of the destination state and a test about pre-conditions it requires.

2. Specification of the Interface for Filling in the Customer's Contact Information

```
"Fill in Customer Contact"
push("Fill in Customer Name",
        test(Monitor),
        action(sendr[Customer],getr[Customer]),
        termin(to "Fill in Service Type")
        )
```

Note that this step involves a pushdown to a subsidiary network. This is so because every single piece of information required from the user is actually going to be added to the same web page (see that the index of the web page under construction remains the same until there is a "pop" arc at the end of the subsidiary network). This process must be monitored by the caller network. The register needed by the called network is passed down to it (actually the Customer information unit).

3. Specification of the Interface for Filling in the Customer's Name

```
"Fill in Customer Name"
cat(Data_Entry,
     test(Validate_Input),
     (
     action(setr 'Interface' getdb[table: 'Customer'; field: 'CustContName']),
     action(setr 'Webpage02'  generate-htmlform[Interface]),
     action(setr 'Webpage02'  generate-htmltext[CUSTOMER_OO.Dept])
     action(setr[CustContName],form_input),
     action(setr[Customer],buildq[CUSTOMER_OO.Name, getr[CustContName]]),
     )
     terminate(to "Fill in Customer Address", post_test)
    )
```

This step builds up the first part of the HTML form the user will see. Before building the input field for the customer name, a header with the previously informed department is generated. This demonstrates how registers hold the information shared by different states and interfaces and make it available during transitions. In this example, the register keeps information that will show up as a header of the web page and later on will be saved in the database. It is actually an information about the context in which the end-user is inserted, which is especially useful during hypertext navigation [Schwabe&Rossi&Barbosa]. The main objective is to help the end-user to recognize the current context.

Notice that again the ATN integrates aspects of the Interface Model, for interface construction, and of the Domain Model (the Customer information unit). This permeates the rest of the productions we will describe, so from now on we will only point out more relevant and differentiating aspects of this integration.

4. Specification of the Interface for Filling in the Customer's Address

This step is analogous to step 3. It adds new interface elements to the form under construction, more specifically the Customer's Address input field. The information structure relative to the created Customer instance is being enriched.

5. Specification of the Interface for Filling in the Customer's Email

```
"Fill in Customer Email"
cat(Data_Entry,
    test(Validate_Input),
    (
    action(setr 'Interface getdb[table:interface;field:CustContEmail]),
    action(setr 'Webpage02 generate-htmlform[Interface]),
    action(setr[CustContEmail],form_input),
    action(setr[Customer],buildq[CUSTOMER_OO.Email, getr[CustContEmail]]),
    action(setdb[table:Customer;field:CustDept], getr[CUSTOMER_OO.Dept])
    action(setdb[table:Customer;field:CustContNamel], getr[CUSTOMER_OO.Name])
    action(setdb[table:Customer;field:CustContAddr], getr[CUSTOMER_OO.Address])
    action(setdb[table:Customer;field:CustContEmail], getr[CUSTOMER_OO.Email])
    )
    terminate(to "Consolidate Information", post_test)
    )
```

Note that this interface performs a series of writings in the application's database. The transition from this state is towards a totally abstract state (see below), whose function is solely that of consolidating structures (to be used for information structuring and text generation) and returning control to the caller network. Web page construction functions have been indexed to the same web page, meaning that all this information is captured in one single HTML form. This is consistent with the interface network actually depicted to the user, as shown in Figure 3. It maintains the conceptual unity of the original interface design, in which customer information is all collected in one dialog between client and server.

6.Specification of an abstract (hidden) interface between two interface networks

```
"Consolidate Customer Information"
pop(  buildq[CUSTOMER_INFO], getr[CUSTOMER],
      test(monitor)  )
tst('error-handler',
    test('Validate_Input'),
    (
        action(setr 'Error  not-consolidate)
    )
    terminate(to 'handle-error', post_test)
    )
```

In this transition we illustrate the general idea of error-handling in X-WITS. If the 'pop' arc cannot be followed, the 'tst' arc is. In it, a special error-handling routine is assigned to the Error register. A transition to an error-handling network is made. At this point an explanation about the error should be generated and displayed to the end-user, so that s/he can try to work out the problem encountered. As already mentioned, exception handling mechanisms will be further detailed in future work.

7. Specification of the Interface for Filling in the Service Type

```
"Fill in Service Type"
cat(Data_Entry,
     test(Validate_Input),
     (
     action(setr 'Interface getdb[table:interface; field:ServType]),
     action(setr 'Webpage03 generate_htmlform[Interface]),
     action(setr 'Webpage03 generate_navigational_anchor[Interface, Customer_OO]),
     action(setr[ServType],form_input),
     action(setr[Service],buildq[SERVICE_OO.Type, getr[ServType]]),
     )
     terminate(to "Fill in Service Data", post_test)
    )
```

Notice that now the index of the web page corresponding to the interface in Qualitas has been incremented. This is the next node after the 'pop' arc of the embedded network is traversed, and the post-test is successfully performed form "Fill in Customer Contact". The rest of the network depicted in Figure 3 involves another push/pop pair of arcs, to gather information about the Service being requested.

The fact that the ATN is a processor whose computational power is equivalent to a Turing Machine is essential to our solution, which can be regarded as the specification of a language processor or a grammar. Its main characteristic is the possibility to make computations during the transition between one state and the next, having that the result of these computations may lead to one state or another. In the task oriented domains we intend to support, this feature is required, as exemplified in the previous ATN instance.

More importantly, the grammar we need has to support both generative and interpretive processes. As an interpreter the ATN is responsible for validating the new workflows developed by the end-users. In summary, this means that a workflow is actually a valid “text” of the ATN grammar, which is read by the computer and end-user during execution. Our EUP solution can be defined as the activity of writing new texts within this grammar. As an ATN, it allows for a possibly infinite number of new texts, a characteristic that presents some level of flexibility to the end-user who is developing the extension.

Generative power is also required, since we plan for automatic generation of documentation (such as a NL workflow documentation and help system) and automatic generation of the extension environment interface, among other resources that could be generated as a derivation or translation of this ATN. This way, the ATN acts as a base language, upon which translations to different languages (such as a subset of NL or a graphical language) can be made.

Our semiotic studies define principles that should be followed when performing such translations. This is one of the topics in our discussion in the next section. The objective is to reach for the usability and applicability of the extensible applications.

## 4. Discussion

This section discusses our proposal for X-WITS based on Semiotics. Later on it relates our proposal to existing End-User Programming research.

### 4.1 Semiotics and Computational Semiotics

Studies in Semiotics and Computational Semiotics guide the development of this work and present different kinds of contributions. As presented in the introduction, they help to understand and reinforce the need for an EUP environment, thus motivating the development of extensible applications.

A cardinal tenet in Semiotic Engineering [de Souza'93] is that a software application is a message that is sent from its designer to the intended end-users. The main content of this message is the designer's understanding of what the domain tasks are and how these tasks should be performed by the end-users. This approach distinguishes our work from Freeflow [Dourish et al], in which the user can choose between different strategies to achieve a certain goal (forward or backward chaining). While in Freeflow an end-user chooses among these strategies at runtime, our proposal is to present to an end-user the strategies which the designer found suitable, but if they are not satisfactory, the user can build his/her own strategy at the extension environment. This distinction favors different approaches to user-interface design. Our work favors a pre-defined step-by-step approach to problem solving and a user-interface design that guides the end-users, such as *wizards*. On the other hand, the proposal of [Dourish et al] implies a "do-it-yourself" approach to problem solving and a more Object Oriented interface in which the end-user decides what should be the next step.

The designer's message reaches end-users through the User Interface Language (UIL) for example. In extensible applications, the EUP language (EUPL) also expresses the designer's message. But the EUPL also enables end-users to manipulate the original designer's message in order to create their own messages. As previously mentioned, extensions can be viewed as an indirect communication or meaning negotiation between end-users and the application designer.

In order to enable this indirect communication two main features are crucial: a shared and relatively stable space of (primitive) meanings and a language in which to articulate the meanings (primitive as well as new meanings). Primitive meanings serve as the reference from which new meanings are introduced. In our proposal, the primitive meanings are established in the original designer message. The ATN is the language in which these meanings are expressed.

As a natural consequence, we have that primitive meanings should not be replaced during extensions (e. g. extensions don't replace the original designer message). Also we have that it should not be possible to create totally new meanings, non related at all to preexistent ones, as new meanings are created when in opposition to existent ones [Eco'76]. Although these considerations lead to restrictions that are imposed on the set of possible extensions, they are the guarantee that the indirect communication between designer and end-users will be possible along time. In future work we will discuss that this is also indispensable for extensions that are targeted at other end-users, since their common ground of understanding is also the original designer's message.

The ATN is the language that enables the indirect communication to occur, be it from designers to end-users or vice-versa. But end-users should not be expected to directly express themselves in

this language. This is why we provide abstractions of this language to our end-users, such as a user interface language and the proposed extension environment.

[de Souza] has proposed principles that describe the relationships that must hold between the different languages involved in the development and usage of an extensible application. In our proposal we applied these principles to guide us during the translation of the ATN into the more abstract languages. A consequence of applying these principles is the enhancement of usability and applicability of extensible software.

Now we describe how our proposal follows these principles. The first principle is called the Interpretive Abstraction Principle. It relies on two definitions: "

1. a lexical sign is a meaningful word of a given language; and
2. a phrasal sign is a meaningful grammatical construct of a given language.

Given any two computer languages Li and Lj, Li is an interpretive abstraction of Lj if:

- the semantics of Li can be described in Lj; and
- a user of Li can make sense of all Li lexical and phrasal signs by resorting to at most three fundamental and possibly intersecting sources:
  i. the intrinsic patterns of occurrence of such signs in situated Li discourse;
  ii. some extrinsic metalinguistic knowledge about Li; and
  iii. the user's cultural background." [de Souza]

In our proposal, we have that the ATN plays the role of Lj. An interpretive abstraction of it is the language proposed in the extension environment, or the EUP language (EUPL). The EUPL also plays the role of Lj, when related to the UIL. Note that our EUPL is used to describe the semantics of User Interface Language (UIL), which is at a higher level of interpretive abstraction. The EUPL actually describes objects users should be able to recognize (e. g. the user interface) and so end-users are capable of resorting on knowledge about the UIL in order to build extensions. This characterizes our interface-driven approach to End-User Programming and in some way enables us to verify that, by following this principle, we potentially enhance usability aspects of our application.

This principle also says that end-users don't have to understand the less abstract languages in order to interact with the UIL and that users should be able to interact with the UIL by resorting on items i, ii and iii listed before. This allows users to focus on information relevant to the domain tasks at hand. Recursively, when interacting with the extension environment, end-users are not required to acknowledge the underlying ATN. But they are driven to learn the EUPL (a less abstract language then the UIL). At the EUPL details hidden at the UIL rise up, such as the interfaces' components (fields and their relationships with the information model, as in figure 8 and 13) and the actions that take place between one interface and the next (as in figure 9). End-users are then driven to learn how to interact with this environment (e.g. learn a less abstract language). Other possible interpretive abstractions of the ATN, such as automatically generated documentation or help system (referred to in item ii) should help the user to make this passage from the UIL to the EUPL.

We can practice this based on the ATN that describes figure 3. The on-line documentation about the "Write Draft" activity can be an interpretive abstraction of the corresponding ATN into a subset of natural language, such as:

*The customer department should be informed, then the customer contact data should be provided. At the next page you should select the type of the service and then inform the rest of service data. After this, all the information provided will be stored in Qualitas' database and you will be asked to transform the draft into a proposal.*

The same applies to error messages (e.g. as result of failed test operations), or to our exception handling mechanism. This mechanism provides an explanation to the user about what has been done before the ATN halted and at what point it halted. This explanation can also be an interpretive abstraction of the unfinished ATN into pseudo-natural language. Graphics depicting interface networks should also be interpretive abstractions of possible ATN instances.

By having the ATN as the base language for different interpretive abstractions, we maximize the chances of success, as the user deals with the more abstract languages. This way different languages have a common point of reference and are likely to maintain some level of co-referentiality [Draper] among them. This potentially enables end-users to recognize the common patterns and meanings among the different languages they deal with.

The other principle is that of Semiotic Continuum. It relies on two additional definitions: "
- An actual sign in a language is an existing word or phrase of that language; and
- A potential sign in a language is a non-existing word or phrase in that language, which can nevertheless be generated by lexical and/or grammatical extensions that abide to its derivational patterns (i. e. to morphological and grammatical meta-rules)."

The principle states that, "given two computer languages Li and Lj, they are said to be semiotically continuous if:
- Li is an interpretive abstraction of Lj;
  - Li is itself an interpretive abstraction of some other language(s);
  - Lj can generate instances of <text>, a specific structured syntactic organization of sentences (whose meaning incorporates intentional elements); and
  - any user who knows Li and Lj can always translate an arbitrary instance of Lj <text> into actual or potential valid combinations of Li signs.

... The most important point in the above definition is that if somebody knows both languages, he or she can always translate any arbitrary **text** of the lower level language into an actual or potential sign of the upper-level one." [de Souza]

By following this principle, a tight coupling between the EUPL and the UIL is achieved. It represents a challenge for EUP languages since, for example, users should be prevented from writing a syntactically correct but useless code, with no effect at the UIL level.

The principle also deals with the case of execution errors, since it requires that texts in the EUPL incorporate error-handling constituents that can be mapped to perceived UIL signs. Since end-users usually don't program for errors, only the tight coupling between the EUPL and the UIL can guarantee that the errors generated by the EUPL may be expressed in UIL terms, in order to help the user to understand what happened.

Since our proposal of end-user extension is interface-driven, it is a fact that every extension developed by the end-user is translated into a potential sign of the UIL (the upper-level language). In terms of error handling ability, we propose that every time an error occurs a NL text should be generated, explaining what has been processed before the error and at what point in the ATN the

processing stopped. An alternative, and possibly complementary, solution would be to take the part of ATN that was processed and translate it into the actual dynamic sequence of interfaces, and display it to the end-user (such as a Lotus ScreenCam movie). These topics will be further detailed in future work about the exception handling mechanisms of X-WITS.

The principles describe the relationships that must hold among the languages that constitute an extensible application, helping application designers to get their messages across to the end user community in different and consistent manners. A contribution of our work to the development of research in Semiotic Engineering [de Souza'93] is the application of these principles in the construction of our proposal for X-WITS. The future implementation of the end-user extensible environment and tests will allow us to achieve more concrete conclusions about the impact of applying these principles.

#### 4.2 End-User Programming

End-User Programming techniques were categorized by [Cypher]. Four categories of techniques were described:

- Parameter Configuration: the user is presented a set of parameters which values may be changed and, as effect, the application will change.
- Macro Recording: the user records a set of steps realized in the User-Interface. These steps are recorded under a label and later on the user can use this label to execute the sequence of steps associated with it.
- Programming by Demonstration: this category is similar to Macro Recording. The difference remains in the fact that this technique tries to overcome the problem of literally interpreting the steps that are being recorded. For example if a step of saving a file called "Paper.txt" is recorded in Macro Recording, in Programming by Demonstration this step could be recorded as "*.txt", where * stands for the name of the current file. A repetitive task in Macro Recording would be saved as a sequence of the same task various times. In Programming by Demonstration, a loop command would be generated.
- Programming Languages: the extensible application offers a programming language to the end-user.

It is important to note that the Parameter Configuration and the Programming by Demonstration techniques present limitations when analyzed from a semiotic perspective. In the Parameter Configuration technique, the designer tries to foresee the extensions users will want to do (in order to provide the set of parameters and their values), which is very hard or even impossible to achieve given the definition of the semiosis concept, that states that it may be unlimited. The Programming by Demonstration falls into the same problem, with the difference that what the designer is trying to anticipate is the generalization the user is looking for.

The Programming by Demonstration and the Macro Recording techniques present as an advantage the fact that end-users extend the application through the User-Interface Language (UIL), building up on this knowledge in order to build extensions. Although the Macro Recording technique presents the problem of literally recording each step, thus limiting the reuse of the macro in different situations or contexts.

Providing a Programming Language to the end-users overcomes the problems previously mentioned, but raises others. The main problem is that users should not be expected to know how to program. So how they could be enabled to manipulate variables, conditions and iterations, in

sum, to use a programming language? Advances are taking place in this area. Scripting languages can be a solution, in which end-users program by using parts of the domain jargon [Nardi]. [DiGiano] proposes a gentle introduction to the end-user programming language, mainly by showing its connections with the UIL at the UIL itself.

Since the UIL is the language used to transmit the designer's message and is the language that users know, usability of the extension environment can be enhanced when end-users extend the application building up on this knowledge. This has been proposed in EUP paradigms such as Macro Recording and Programming by Demonstration, but the disadvantages described come along as well.

Ideally users should be provided with the power of a programming language and at the same time be able to reuse the knowledge they acquired about the UIL during the extension process. [DiGiano] proposes the exposition of the EUPL in connection with the UIL at the UIL. Our proposal goes the other way around, and turns the user interface as the basis upon which EUP takes place. As a consequence, the meanings conveyed through the UIL, in addition to the ones conveyed in the extension environment, (the designer message) are the programming building blocks provided to end-users.

In Oval [Malone&Lai&Fry] these building blocks are namely Objects, Views, Agents and Links. Its proposal is to offer end-users a radically tailorable tool that can be used to construct a large range of applications for cooperative work. The tool reaches its goal in the sense that various kinds of applications can be constructed, especially due to the generality of the primitive building blocks. On the other hand, as mentioned by the authors, a key issue arises when users have to share their extensions with other users in the cooperative environment, for instance, when users create their own objects and then have to share this information (e. g. meanings) with other users. The common and stable ground of meanings shared by the end-users rests in the very abstract, domain independent level of the primitive building blocks. The authors then argue that some tailoring features (such as the definition of new object types), or maybe the creation of new and complete applications, may be done by experienced end-users (e. g. power users or programmers). They conclude that Oval provides a wide spectrum of tailoring options and that many users would only make changes to applications developed by others.

This is the level that our proposal fits in. As in [Nardi], we believe that end-users should be able to extend an application by resorting on the domain jargon they already know, and not in the generic building blocks proposed by Oval. Our ATN (and all the languages derived from it) establishes and expresses the common set of meanings that are shared by the community of end-users and that are used throughout the application. More than that, we propose the extension mechanisms that impose restrictions on the set of possible extensions in order to try to preserve these meanings.

The main contribution of our proposal to EUP remains on the common representation language from which all the other application languages are derived. This enables the content of designer message about the task oriented domain to be shared the among the different languages in a consistent manner. More than that, we discuss that the original message should not be directly altered by the end-users, since it represents the common point of reference between end-users and designers. Another contribution is our proposal of how the user interface could be used in the extension environment, acting as building blocks that end-users can manipulate in order to build the desired extensions.

## 5. Conclusion

We have presented the main aspects of a model that supports eXtensible, Web based, Information intensive, Task Oriented Systems (X-WITS) and described an extension scenario in which it could be applied. X-WITS help in some way the delivery of flexible workflow applications. Flexibility by selection [Heinl et al] offers multiple alternative execution paths to the end-users and can be provided by the application designer. But, as stated by the authors, "flexibility by selection is limited by the requirement that it has to be anticipated and has to be included (directly or indirectly) into the workflow type specification". The semiosis concept as a background prepares for the need to handle unpredicted situations or user needs. So, the proposed extension environment provides some level of flexibility by adaption, in which an unforeseen execution path can be incorporated. Workflow adaptions may take place on one or more specific running workflow instances or deliver a new workflow path that does not affect the running instances. In our proposal, only variants of the current workflow are generated – the end-user extensions coexist with the designer's workflow and are not new workflow versions that replace the current one. It is important to emphasize that the possible adaptations are restricted, so that they are potentially consistent with the original designer's message.

The strength of this proposal relies on the described representation language - the ATN. It represents the designer's message and serves as a common ground that is used for the indirect communications mentioned: from the designer to end-user (e. g. in the UIL), from the end-user back to the designer (when developing extensions) and from one end-user to another (when developing extensions targeted at other users). We discuss why this common ground should be preserved throughout application usage and reflect this in the described example.

It is possible to derive different languages from the ATN. By deriving many languages from the ATN, always following the principles proposed by [de Souza], the chances of successfully getting the designer's message across is increased and thus application usability and applicability should be achieved.

As future work we plan for the further specification of the proposed models and representation language and then the implementation and validation of this proposal. We also foresee that a set of different directions can be followed from now on. The discussions about exception handling and EUP in groupware systems should be further explored. Note that our proposal relates to two different axis of groupware, as in [Malone&Lai&Fry]. One axis is groupware involving different end-users of the same application (such as the technicians and coordinators of the exemplified domain). The other axis is groupware involving the end-users that extend the application and the application designers, in the sense that they should all work together in order to accomplish certain kinds of extensions. We also intend to evaluate how analogic mechanisms (e. g. metaphors) could be used during extension, as proposed by [Barbosa, Barbosa&de Souza'99, Barbosa&de Souza'2000], in order to enhance usability of the extension environment.

It will also be interesting to see how EUP can be possible in an application that has pseudo-natural language and hypertext automatic generation, so that these parts also work for the extensions developed by the end- users.

## 6. References


[Abbott&Sarin] Abbott, K. R. and Sarin, S. K.; 1994; Experiences with Workflow Management: Issues for the Next Generation; Computer Supported Cooperative Work'94 (CSCW'94); pages 113-120; ACM

[Barbosa] Barbosa, S. D. J.; 1999; Programação Via Interface; Doctorate Thesis; Advisor: Clarisse Sieckenius de Souza; PUC-Rio

[Barbosa&de Souza'99] Barbosa, S. D. J. and de Souza, C. S.; 1999; Making More Sense out of Users' Utterances; II Workshop sobre Fatores Humanos em Sistemas Computacionais - IHC'99 Proceedings

[Barbosa&de Souza'2000] Barbosa, S. D. J. and de Souza, C. S.; forthcoming (2000); Extending software through metaphors and metonymies; To be published at IUI'2000 (Intelligent User Interfaces'2000). New Orleans, LA. January, 2000.

[Cypher] Cypher, A. (ed.); 1993; Watch What I do: Programming by Demonstration; The MIT Press

[de Souza'93] de Souza, C. S.; 1993; The Semiotic Engineering of User Interface Languages; International Journal of Man-Machine Studies; No. 39; pages 753-773

[de Souza'97] de Souza, C. S.; 1997; Semiotic Engineering Principles for Evaluating End-User Programming Environments; Interacting with Computers; Elsevier

[de Souza&Barbosa] de Souza, C. S., Barbosa, S. D. J.; 1996; End-User Programming Environments: The Semiotic Challenges; Lucena, C. J. P. (ed.) Monografias da Ciência da Computação; Departamento de Informática. PUC-RioInf MCC 16/96.

[de Souza&Dahis] de Souza, C. S., Dahis, G.; *forthcoming*

[Chiu&Karlapalem&Li] Chiu, D. K. W., Karlapalem, K. and Li, Q.; 1998; Exception Handling with Workflow Evolution in ADOME-WFMS: a Taxonomy and Resolution Techniques; Computer Supported Cooperative Work'98 Workshop (CSCW-98 Workshop) - Towards Adaptive Workflow Systems

[DiGiano] DiGiano, C.; 1996; A vision of highly-learnable end-user programming languages; Child's Play'96; Position Paper

[Dourish et al] Dourish, P., Holmes, J., MacLean, A., Marqvardsen, P. and Zbyslaw, A; 1996; Computer Supported Cooperative Work'96 (CSCW'96); pages 190-198; ACM

[Draper] Draper, S. A.; 1986; Display Managers as the Basis for User-Machine Communication; In Norman and Draper (eds.) User-Centered System Design; Lawrence Erlbaum and Associates

[Eco'76] Eco, U.; 1976; A Theory of Semiotics; Indiana University Press

[Eco'84] Eco, U.; 1984; Semiotics and the Philosophy of Language; Indiana University Press.

[Fischer] Fischer, G.; 1998; Beyond 'Couch Potatoes': From Customers to Designers; Proceedings of the 5th Asia Pacific Computer Huma Interaction Conference; IEEE Computer Society; pp. 2-9

[Glance et. al.] Glance, N. S. and Pagani, D. S. and Pareschi, R.; 1996; Generalized Process Structure Grammars (GPSG) for Flexible Representations of Work; Computer Supported Cooperative Work'96 (CSCW'96); pages 180-189; ACM

[Heinl et al] Heinl, P., Horn, S., Jablonski, S., Neeb, J., Stein, K. and Teschke, M.; 1999; A Comprehensive Approach to Flexibility in Workflow Management Systems; International Joint Conference on Work Activities Coordination and Collaboration - WACC'99; ACM

[Malone&Lai&Fry] Malone, T. W. and Lai, K. and Fry, C.; 1995; Experiments with Oval: A Radically Tailorable Tool for Cooperative Work; ACM Transanctions on Information Systems; Vol. 13; No. 2; April 1995; pages 177-205

[Myers] Myers, B. A.; 1992; Languages for Developing User Interfaces; Jones and Bartlett Publishers, Inc. Boston.

[Nardi] Nardi, B.; 1993; A Small Matter of Programming; The MIT Press

[Peirce] Peirce, C. S.; 1931; Collected Papers; Harvard University Press.

[Prates] Prates, R. O. and de Souza, C.S.; 1998; On the Rationale of Interface Semiotics for Multi-User Applications; ISIC/CIRA/ISAS'98 A Joint Conference on the Science and Technology of Intelligent Systems; IEEE and NIST

[Schwabe&Rossi&Barbosa] Schwabe, D. and Rossi, G., Barbosa, S. D. J.; 1996; Systematic Hypermedia Application Design with OOHDM; Hypertext'96 - Seventh ACM Conference on Hypertext (http://www.acm.org/siglink/ht96/)

[WMC] Workflow Management Coalition – Terminology and Glossary; Document Number WFMC-TC-1011, Document Status – Issue 3.0; 1999.

[WMC'1995] Workflow Management Coalition – The Workflow Reference Model; Document Number WFMC-TC00-1003, Document Status – Issue 1.1; Hollingsworth, D.; 1995.

[Woods] Woods, W. A.; 1970; Transition Network Grammars for Natural Language Analysis; Communications of the ACM; Vol. 13; No. 10; pages 591-606.