Form Management in the Vicomte Workflow System *

Daniel K.C. Chan †

Project VERSO, INRIA - Rocquencourt, Domaine de Voluceau,
B.P. 105, 78153 Le Chesnay Cedex, France
Email: Daniel.Chan@inria.fr, Tel: +33 1 39 63 52 66, Fax: +33 1 39 63 56 74

Abstract

It is one of the distinguishing characteristics of workflow management systems to integrate automated processes with human processes in an application system. The way a human process is modelled in a workflow system is by capturing the significant events of the process. These events include notifying the user who is assigned to perform the process, disseminating information that is necessary for the user to carry out the process, and collecting information that is generated as the result of the user process. Such events are often presented to the user or captured from the user using forms. This paper presents a form management system that is part of the Vicomte workflow management system. The major components include a form specification language, a facility for dynamically generating HTML pages with client-side validation to be used over the internet, as well as processing modules for the generated HTML pages on the server side. The architecture of the form management system is rather general and therefore is applicable for many systems requiring an effective form processing facility.

1 Introduction

Workflow management systems are attracting a lot of attention partly due to its ability to overcome some of the problems in existing systems concerning integration in various aspects. Normally, an application system captures only automated processes without much concern of user processes that cannot be automated. Workflow management systems attempt to capture some aspects of human processes by addressing the problem of task assignment as well as information dissemination and collection. In a nutshell, workflow management systems are extending the scope of process modelling to include some information of human processes so as to help with scheduling, control, and analysis of individual activities as well as the workflow application as a whole.

The way a human process is modelled in a workflow system is by capturing the significant events of the process. These events include notifying the user who is assigned to perform the process, disseminating information that is necessary for the user to carry out the process, and collecting information that is generated as the result of the user process. Such events are often presented to the user or captured from the user using *forms*. A form can display information to the user who is then expected to provide some input which may be decision information or data obtained from the process that has been carried out. Such information often affects the execution of the rest of the workflow application.

This paper presents a form management system that is part of the *Vicomte* workflow management system. *Vicomte* uses HTML forms as the primary means of interaction with users who may be located in different places and move from place to place. Since it relies on the internet as its communication infrastructure, the possibility of distribution and the freedom of mobility are guaranteed. The major components of the form management system include a *form specification language*, a facility for dynamically generating HTML pages with client side validation, as well as the server side processing modules for the generated HTML pages. Since it is often the case that users would like to add a few finishing touches to the system generated HTML pages, *Vicomte* allows users to do so by maintaining the generated HTML pages in files managed by the native operating system. Dy-

*Copyright 1999 IEEE. Published in the Proceedings of the Hawaii International Conference on System Sciences, January 5-8, 1999, Maui, Hawaii. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

†Supported by a Marie Curie Fellowship from the European Commission (ERBFMBICT960653).

namic information are then added when the HTML pages are to be dispatched to users. It should be noted that such flexibility is not often entertained by systems generating dynamic HTML pages. Actually, most workflow systems do not support form management or leave the responsibility to tools, such as Microsoft Front Page [2], as in METEOR [19, 6], that are not integrated with the workflow systems at all. The system described in this paper therefore provides a bridge for this gap. Moreover, it also allows users to redesign the generated HTML pages if they wish. The system architecture is rather general and can be applied with minimal changes to many systems requiring an effective form processing facility.

The rest of the paper is structured as follows. Section 2 describes related work about form management. Section 3 introduces the *Liaison* workflow model, which is the underlying workflow model in the *Vicomte* workflow system. Section 4 presents the form specification language using an example application. Section 5 examines the architecture of the system. Section 6 discusses the design choices and the implementation of the system. Section 7 concludes.

2 Related Work

FORMANAGER [20] was developed as an office information system primarily for the manipulation and routing of data stored in relational database systems. Forms are defined interactively. In addition to textual information, a form can contain a number of fields. A field can be a search field providing a parameter for selecting data from the database, a display field presenting data retrieved from the database, or an entry field allowing data to be displayed and changed. Conditions involving relational operators (such as ">="), database relations, and their attributes can be used in search fields. A display field can be associated to either an attribute of a relation or an expression. Actually, it is possible to select data from more than one relation by putting a number of conditions involving different relations in a search field or a display field. Moreover, a field can be associated with an action which specifies how the field can be used to update the database. The basic routing capability of FORMANAGER is reminiscent of the process modelling capability of today's workflow management systems. As far as form management is concerned, FORMANAGER and *Vicomte* are built on rather different philosophies. It is the emphasis of *Vicomte* to separate the presentation of data as well as the analysis of data in a form from the underlying storage mechanism. The tight coupling approach adopted in FORMANAGER makes it very vulnerable to changes at the database level. (For an alternative loose coupling approach between databases and data representation, interested readers are referred to the proposal given in [3].) There is no mechanism to capture the relationships between fields in the same form which makes analysis impossible.

Like FORMANAGER, OPAS [12] aimed to provide a framework for office information system in which processes are modelled with forms as input and output. Unlike FORMANAGER, there is no distinction between forms and data. As an example for the later, a relation would be modelled as a *form file* with a number of *form instances* in it. Consequently, there are notions, such as *field uniqueness*, that make sense at the data level but seem rather unnatural at the form level. Form operations, such as *union*, are provided and again they operate really only at the data level. Richer selection conditions are allowed, for example, "IS NOT NULL".

OFS (Office Form System) [18] was built with the aim to generalise office procedures by abstracting office documents of various media into data and their mapping to template of various media as well as the routing of these forms from one machine to another. Forms can be duplicated and consistency between copies is maintained. Form templates support range checking for fields and the "NOT NULL" post-condition. It also emphasises on the use of trace log to record changes to fields and the machine on which the fields are changed.

There are other forms systems developed with more or less the same goal. On one end of the spectrum there is FADS [11], which is completely database-oriented with form templates mirror the data structures of the underlying relational database. On the other end of the spectrum is Star [10], which is positioned as an office information system for activities that are not less well structured. Star emphasises the principle of "no programming, only point and click" and hence a uniform interface is the primary concern. Data are defined via the field definitions given in forms instead of the other way round as found in other systems and are stored in files rather than databases. It is a simple form management system when compared to the others.

On the whole, all the above-mentioned systems lack the pre-condition, the post-condition, and form derivation supported in *Vicomte*. Since these systems were developed at a time when technologies are rather different from what is common nowadays, it is not surprising that *Vicomte* suits better for today's technologies than the systems examined in this section.

3 An Overview of Liaison

The underlying workflow model used in *Vicomte* is called *Liaison*. This model is an extension of the reference workflow model proposed by the Workflow Management Coalition [7]. Like the reference workflow model, it captures the fundamental elements of the workflow paradigm: organisation model, information model, process model, and their relationships. Unlike the reference workflow model, it supports a rich organisation model, sophisticated activity assignment constraints, and a *form model* concerning the presentation and access control of data. The latter and its implementation is the subject of this paper. The example used in this paper is expressed using *Valmont* [4], which is a textual representation language of *Liaison*. A summary of the "sub-models" of *Liaison* is given next. More information about *Liaison* can be found in [4].

3.1 Organisation Modelling

The very first stage of every application development project would be to approach the personnel related to and affected by the project so as to understand the general objectives, requirements, and operation environment. This often translates to figuring out the organisation hierarchy and interviewing senior personnel before interviewing less senior personnel. The organisation hierarchy can be captured as a position hierarchy.

It should be noted that such a hierarchy captures the delegation and accountability relationships but does not usually reflect the functional and operational structures of an enterprise. However, as far as workflow application design is concerned, in many cases, the operational structure of an activity can be constructed out of the position hierarchy. In order to maximise the reusability of the position hierarchy and to minimise the effort required to model the operational structures, an elaboration of the lower levels of the position hierarchy is encouraged. Relatively speaking, a less refined account of the upper levels would suffice.

The functional structures of an enterprise can be captured using teams and the team hierarchy. The grouping of individuals into teams should also take activity assignment into consideration. For example, a large team would be preferred for high volume or highly repetitive activities. It should be noted that it is possible and likely that the team hierarchy contains free standing teams that are not be related to any other teams (see the team hierarchy on the right of the abstract organisation structure shown in Figure 1).

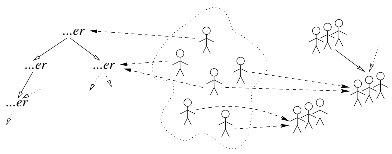

Figure 1. Positions, Staff, and Teams

The relationships between positions, individuals, and teams are shown in Figure 1. The position hierarchy is shown on the left where a *...er* could stand for a position such as Information Technology Manag*er* or Programm*er*. The assignment of individuals to positions and to teams is shown by the dotted line arrows.

3.2 Information Modelling

The information model defines data used in a workflow process, governs operations that can be performed on the data, as well as controls the presentation of data. In other words, the model defines data that are passed between activities within a workflow process as well as data that go across the boundary of the workflow process. The first kind of data is referred to as *local* data which can be application data or control data. They are managed by the workflow management system. The second kind of data is referred to as *external* data which serve as connections between the workflow process and other external systems that may be heterogeneous and distributed in different geographical locations. For instance, a system design process may need to access a database containing templates for system testing, it may also invoke another application for the purpose of processing the functional specification, or it may need to communicate with an existing workflow process dealing with the interview procedure as shown in Figure 2. A piece of such external data is represented in *Valmont* as an *entity* whose state may not be known to the workflow systems but usually operations are defined to manipulate the data. Both local and external data are defined in the *data model* which is a sub-model of the information model.

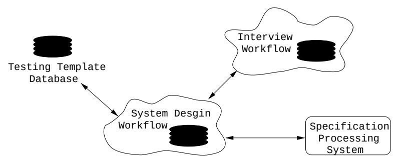

Figure 2. Data Access of a Workflow Process

Presentation of data is defined independently in another sub-model called the *form model*. A form is a unit of presentation that is used whenever interaction with the user is required. It is possible that several forms can be displayed for a single task. Typically a form has a number of fields in which data can be displayed and captured. To improve quality of the display, formatting information can be provided. One form can be derived from another form possibly adding new fields and restricting operations that can be applied on some fields. This mechanism encourages re-usability which, in turn, reduces the maintenance effort. The *form model* is discussed in detail in the next section and thereafter.

3.3 Process Modelling

The *process model* plays a central role among the sub-models of *Liaison*. It ties the organisation, data, and form models to activities representing the steps of the application being modelled. To facilitate the modelling of complex systems as well as making the resultant design more comprehensible, the process model supports incremental modelling allowing an activity to be refined into more detailed activities (see Figure 3). In other words a typical design would end up with several layers of activities with more abstract activities called *tasks* at the upper layers and more concrete activities called *base tasks* appearing in larger number at the very bottom layer.

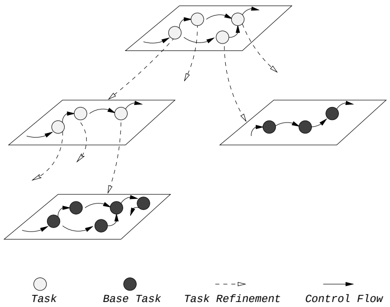

Figure 3. Task Refinement

The concern of the first stage of workflow process modelling is to identify the concerned agents and figure out the activities they have to carry out to fulfill the application. The identified agents should have been captured in the organisation model. Activities and their order of execution (i.e. *control flows*) together with the responsible agents (i.e. *activity assignment*) are then captured in the process model. Abnormal situations in the process can then be identified as *exceptions* which may be triggered by activity events that happen in the process as well as temporal events that are defined in the process. For instance, it is possible to raise an exception when no agent can be found to carry out an activity.

So far what has been discussed can be applied to both tasks and base tasks; however, base tasks can perform other actions that are not found in tasks. The primary difference is the possibility of displaying forms (described in the form model) to initiate some interaction with the user. It is not unusual that the data and form models need to be revised during process modelling.

4 Form Specification Language

4.1 Example Application

This section introduces a form specification language using an example taken from an English insurance company. The application deals with on-line car insurance quotation request. A potential customer can request for a quotation of car insurance by filling in a form that can be accessed via the internet. When the form is filled and processed, a quotation will be returned to the potential customer together with the information supplied in the request form.

A form model with two form definitions is defined in Figure 4. The two forms are significantly simplified and retain only about one-fifth of the information found in the original forms used in the application. Having said that they do include all the essential features of the form specification language.

```
FORM_MODEL Insurance_Quotation

  FORM Quotation_Request

    TITLE "Insurance Quotation Request";

    HEADING "Main Driver Details:";
      LABEL "Full name" LINKS STRING name SIZE 40; NEW_LINE;
      LABEL "Address" LINKS STRING address SIZE 120; NEW_LINE;
      SPACE;
      LABEL "House owner?" LINKS BOOLEAN owner = false; NEW_LINE;
      LABEL "Date of birth" LINKS DATE birthdate; NEW_LINE;
      LABEL "Sex" LINKS STRING sex SIZE 1; NEW_LINE;
      LABEL "Years since passing test" LINKS INT experience;

    HEADING "Vehicle Details:";
      LABEL "Model" LINKS STRING model;
      LABEL "Engine size" LINKS INT engine;
      LABEL "Current value" LINKS FLOAT cover;
      LABEL "Estimated annual mileage" LINKS INT mileage;

    INITIALLY mileage = 12000;
    FINALLY all IS NOT null;

  END_FORM

  FORM Quotation DERIVED_FROM Quotation_Request

    TITLE "Insurance Quotation";

    HEADING "Cover Details:";
      LABEL "Full comprehensive" LINKS FLOAT comprehensive;
      LABEL "Third party, theft, and fire" LINKS FLOAT thirdParty;

    THROUGHOUT all READONLY;

  END_FORM

END_MODEL
```

Figure 4. Form Model Definition in Valmont

4.2 Syntax & Semantics

A form resembles a record except that it also contains presentation information which is important due to the interactive nature of workflow systems. Presentation information include headings, field labels, and formatting information. Each field can be used to display or update data variables defined in the data model. Default field values can also be specified. A form can be derived from another form possibly by adding some fields. Moreover a derived form can restrict operations that can be performed on a field. Next, the syntax and semantics of the form specification language is examined.

Each form model has a name (line 1) and can contain a number of form definitions - two in this case (line 2–19 and 20–26). Each form definition is identified by a name (line 2 and 20) and the name must be unique within the form model.

A form begins with one optional title (line 3) which stands for the main heading of the form page. After the "TITLE" clause, a number of blocks of clauses can be given (line 4–11 and 12–16) each of which starts with a "HEADING" clause. Clauses in the same block are formatted together so it is more a physical notion than a logical one. Fields in a form are defined using the "LABEL" clause, which defines the label used for the field, the type of the value to be held in the field, a name for the field, an optional initial value, an optional length restriction for input, and an optional access constraint. For instance, line 5 states that the field is to be given the label "Full name", accepts a string value of no longer than 40 characters, and this value is to be referred to as "name". Initial value can be specified as in line 8. "NEW_LINE" and "SPACE" clauses can be used to adjust the formatting of the fields in a form (line 5 and 7).

Constraints can also be given at the end of the form definition (line 17–18). In other words, initial value and access control constraints can be given either in the "LABEL" clause, as shown before in line 8, or in the "INITIALLY" clause (line 17) or the "THROUGHOUT" clause (line 25). In the case of line 17, it states that the "mileage" field will assume an initial value of 12000. The "FINALLY" clause can only be specified at the end of the form definition to assert a post condition that must be satisfied before the form can be returned for further processing. The keyword "**all**" is a shorthand for all the fields defined in the form and can be used in the "INITIALLY", "THROUGHOUT", and "FINALLY" clauses. Since "**null**" requires a special treatment of equality, two special operators ("IS" and "IS NOT") are therefore introduced to distinguish the special treatment from the normal reading of "==".

The "Quotation" form is derived from the "Quotation_Request" form. As a result of that, "Quotation" contains all the fields defined in "Quotation_Request" together with their initial values, and access constraints as well as the post condition of "Quotation_Request". A derived form can only add information to its parent form with the only exception of the "TITLE" clause which can be replaced by the derived form as shown in line 21. Additional fields can be defined in a derived form. Two fields are added in "Quotation" (line 23 and 24). Moreover, further constraints can be imposed and they are combined with the constraints in the parent form resulting in more restrictive constraints for the derived form. In this case, "Quotation" asserts in line 25 that all fields are to be made "READONLY". Such a constraint is a convenient means of turning an input form into an output form which happens very often, for example, when a confirmation is to be sought from the user. In the case of the insurance application, the potential customer is presented the quotation together with the information he or she supplied in the request form.

4.3 Usage

Forms are used in the process model to initiate interaction with the user. A form is called with a set of parameters which may be constants or data defined in the data model. For example, the "Quotation_Request" form can be activated as follows:

```
VIEW Quotation_Request (  name = c_name,
                          address = c_address,
                          owner = c_owner,
                          birthdate = c_birthdate,
                          sex = c_sex,
                          experience = c_experience,
                          model = car_model,
                          engine = car_engine,
                          cover = car_cover,
                          mileage = car_mileage );
```

This causes the current value of "c_name" - an data item defined in the data model - to be used as the initial value of the "name" field if no initial value is given in the form definition. Similar initialisation occurs to other fields appeared in the parameters. When the form is returned for processing, the value of the "name" field is used to update the value of "c_name".

4.4 Form Manifestation

Forms are realised as HTML pages with embedded JavaScript code to verify the constraints. The manifested "Quotation_Request" and "Quotation" forms when viewed using the Netscape Navigator will look like those in Figure 5 and 6. Further discussion on the translation to HTML pages and the incorporation of validation code is provided in the section on implementation.

5 System Architecture

The system architecture of the form management system is depicted in Figure 7. The system takes as input a *Valmont* specification which usually contains a data model specification and a form model specification. Using information

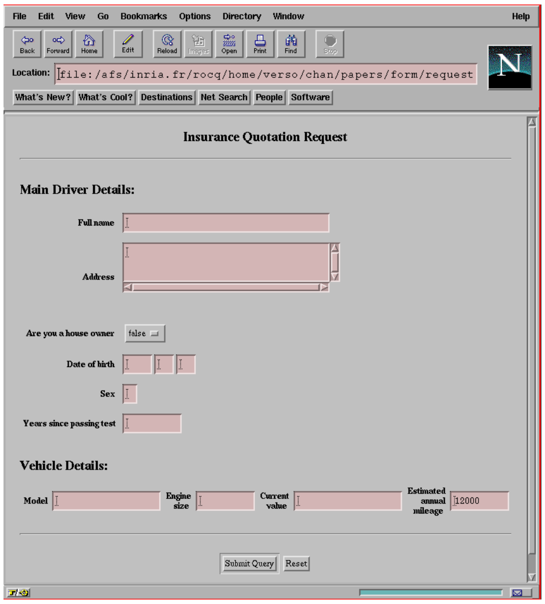

Figure 5. Quotation Request Form

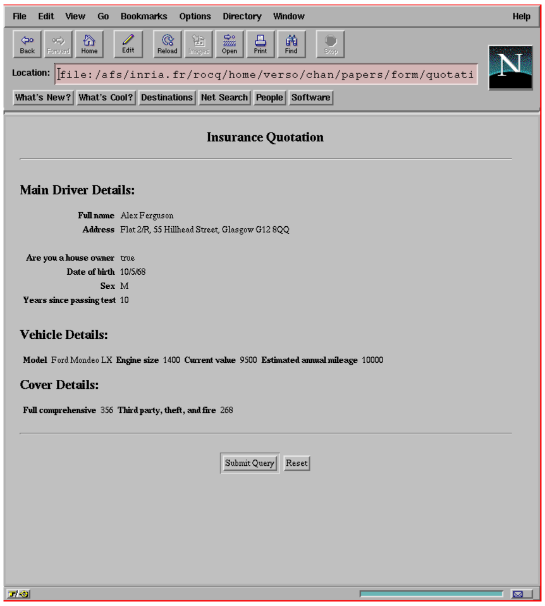

Figure 6. Quotation Form

recorded in the system stores, the system generates HTML pages at run-time so as to allow data to be displayed and new values to be collected.

The round-cornered box represents the form management system of *Vicomte* with three major components in it. The component on the left hand side represents the *build-time* system which is used to define a workflow application. The processing of a *Valmont* specification involves several logical steps. First, the specification is turned into an abstract syntax tree by the *Valmont* parser. This step includes checking on cyclic form definitions and duplicate fields. Second, all derived forms are normalised by expanded their definitions with those of their parent forms. This implies that there is no semantic relationship between a parent form and a child form - it is just a specification reuse mechanism. Third, the specification is type-checked to make sure that all initial values and constraints have the correct types. Fourth, HTML templates are generated. It should be emphasised that these are only logical steps; actually the system has a single-pass parser which does the four steps with a single scan of the specification.

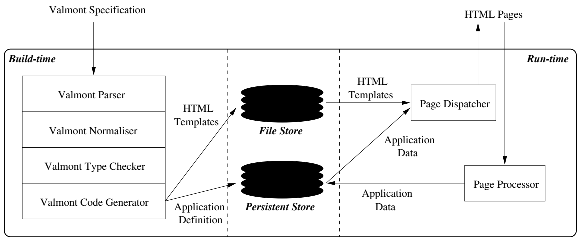

Figure 7. System Architecture

Two storage facilities are provided by the system and they are depicted in the middle column of Figure 7. Each facility has its own manager (not depicted in Figure 7) responsible for storing, changing, and retrieving of information of various sorts. The *persistent store* keeps all the information of a workflow application as well as the data for each instance of the workflow application. The *file store* is used for storing the HTML templates using the native file system provided by the underlying operating system.

The *run-time* system is shown on the right hand side. When an HTML page is to be displayed upon the request of the web server, the corresponding template is located in the file store and filled in with the current values of data from the persistent store before it is forwarded to the server. There is also a *page processor* responsible for responding to the in-coming HTML pages submitted by the user by storing the updated data back into the persistent store. It should be noted that the persistent store does provide concurrency control over simultaneous access of data.

6 Implementation

6.1 Technologies & Tools

The *Vicomte* workflow management system is intended to be run on Windows NT and UNIX Solaris with Java [16] as the primary implementation language. The current implementation is using the Java Development Kit version 1.1.6 [15]. In the case of the form management system, the implementation uses also JavaScript version 1.2 [9] to be used with Netscape Navigator version 4.0 and Microsoft Internet Explorer version 4.0. A number of Java tools are also employed: PJama release 0.4.6.12 - an alternative Java Virtual Machine with support for persistence [1, 17], JavaCC version 0.7.1 - a Java Compiler Compiler [14], and JJTree version 0.3 pre-release 5 - a preprocessor for JavaCC that inserts abstract syntax tree building actions at various places in the JavaCC source [14].

Figure 8 shows the major Java classes developed for the form management system and their relationship. The rest of this section explains the role each of them plays in the implementation.

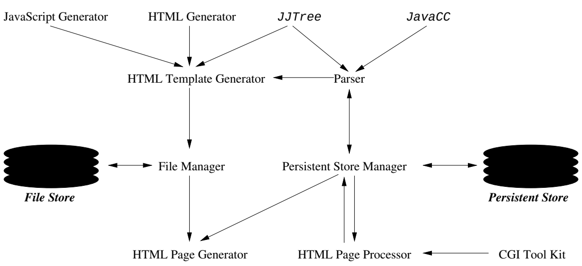

Figure 8. Implementation Modules

The parser is generated using JavaCC which takes a syntactic description of *Valmont* in the form of rules similar to those used in lex & yacc [8]. It is our experience that JavaCC is more flexible than lex & yacc in the sense that directives, such as various kinds of lookahead options, can be added easily at various points in the syntax rules. This helps to preserve the design of the language and avoids the usual practice of distorting the syntax for parsing convenience. We found the error reporting facility very helpful. The openness of the system also makes it possible to control the parsing behavior of the generated parser by directly manipulating the *state* of the generated parser. This is what we did for the normalisation step in the *Valmont* parser. JavaCC becomes even more attractive when used together with JJTree, which helps construct abstract syntax trees.

6.2 Translating to HTML Pages

Besides the use of JJTree and the parser, the translation of the form specification into HTML pages by the "HTML Template Generator" is assisted by two Java classes, namely "HTML Generator" and "JavaScript Generator". We will begin with a discussion on the use of the "HTML Generator" class with the "HTML Template Generator" class. The purpose of having the "HTML Generator" class is to isolate the HTML code from the rest of the generation process. Next, we present the translation scheme using rules in denotational style [13]: using compositional translation functions and the argument to each translation is put within ⟦ ⟧ brackets.

The translation of a form begins with the building of an HTML page framework as illustrated by rule 1. The identifier of the form (I) is translated into the three parts of the HTML page: (1) the title of the HTML page, (2) the name for the object representing the page, and (3) the value of a built-in hidden field called "**Form_ID**" (rule 1). The "TITLE" clause (T) gets translated into a heading as shown in rule 2. The rest of the clauses are translated block by block as governed by rule 3. The constraint C in rule 1 represents the post condition of the form (i.e. a "FINALLY" clause) and does not get translated into HTML but to JavaScript which will be covered in the next subsection. Also noted that one and only one constraint remains as the others (i.e. "INITIALLY" and "THROUGHOUT" clauses) are normalised by the parser before the translation takes place.

```
TF ⟦ "FORM" I T Bs C "END_FORM" ⟧ ⇒
    < HTML >
     < HEAD >
      < TITLE > I < /TITLE >
     < /HEAD >
     < BODY >
    TF ⟦ T ⟧
     < FORM NAME="I"
          ACTION="http://www-rocq.inria.fr/cgi-bin/vicomte"
          METHOD=GET >
     < INPUT TYPE=HIDDEN NAME="Form_ID" VALUE="I" >
    TF ⟦ Bs ⟧
     < /FORM >
     < CENTER >
     < INPUT TYPE=SUBMIT >
     < INPUT TYPE=RESET >
     < /CENTER >
     < /BODY >
     < /HTML >                                          (1)

TF ⟦ "TITLE" S ";" ⟧ ⇒
    < H1 ALIGN=CENTER >  TF ⟦ S ⟧  < /H1 >
    < P >< BR >< P >                                    (2)

TF ⟦ "HEADING" S ";" Es ⟧ ⇒
    < H2 >  TF ⟦ S ⟧  < /H2 >
    < TABLE >
    TF ⟦ Es ⟧
    < /TABLE >                                          (3)
```

The "SPACE" clause has a simple translation as shown in rule 4. The "NEW_LINE" clause serves as a marker to collect together fields that should appear on the same line. In HTML terminology, all clauses that appear before a "NEW_LINE" are put together as a row in a table (rule 5).

```
        TF ⟦ "SPACE" ";" ⟧   ⇒   < P >< BR >< P >       (4)

TF ⟦ Es "NEW_LINE" ";" ⟧   ⇒   < TR >
                               TF ⟦ Es ⟧
                               < /TR >                  (5)
```

Translation of the "LABEL" clause is not as straight forward as the others. For example, "READONLY" fields are translated into simple text rather than input fields. Rule 6 and 7 state that the initial value of a field should be used for the translation whenever possible.

$$\begin{aligned}&\mathbf{TF}\llbracket \text{“LABEL”}\ S\ \text{“LINKS”}\ T\ I\ \text{“=”}\ V\ \text{“SIZE”}\ N\ \text{“READONLY”}\ \text{“;”}\rrbracket \ \Rightarrow\\ &\quad <\text{TH ALIGN=RIGHT}>\ \mathbf{TF}\llbracket S\rrbracket\ </\text{TH}>\\ &\quad <\text{TD}>\ \mathbf{TF}\llbracket V\rrbracket\ </\text{TD}>\end{aligned}\tag{6}$$

$$\begin{aligned}&\mathbf{TF}\llbracket \text{“LABEL”}\ S\ \text{“LINKS”}\ T\ I\ \text{“SIZE”}\ N\ \text{“READONLY”}\ \text{“;”}\rrbracket \ \Rightarrow\\ &\quad <\text{TH ALIGN=RIGHT}>\ \mathbf{TF}\llbracket S\rrbracket\ </\text{TH}>\\ &\quad <\text{TD}>\ \mathbf{TF}\llbracket I\rrbracket\ </\text{TD}>\end{aligned}\tag{7}$$

For an input field, the translation depends on the type of the field. The only common translation for each field is that the label gets translated in the same way as show in rule 8.

$$\begin{aligned}\mathbf{TF}\llbracket \text{“LABEL”}\ S\ K\rrbracket \quad\Rightarrow\quad &<\text{TH ALIGN=RIGHT}>\ \mathbf{TF}\llbracket S\rrbracket\ </\text{TH}>\\ &<\text{TD}>\ \mathbf{TF}\llbracket K\rrbracket\ </\text{TD}>\end{aligned}\tag{8}$$

The translation of an integer data field is governed by rule 9 and 10. The value of N is used to determine the size of the input field. If it is not given in the specification, some default value will be used by the translation scheme. The identifier of the field is translated to the name of an object representing the input field. Like in the case of “READONLY” fields, the initial value is given priority over the current data value for the translation. Field of other data types are translated into “<INPUT>”, “<TEXTAREA>”, “<SELECT>”, and multiple fields in the case of type “DATE”.

$$\begin{aligned}&\mathbf{TF}\llbracket \text{“LINKS”}\ \text{“INT”}\ I\ \text{“SIZE”}\ N\ \text{“;”}\rrbracket \ \Rightarrow\\ &\quad <\text{INPUT TYPE=TEXT SIZE=}N\ \text{MAXLENGTH=}N\ \text{NAME=“}I\text{”}\\ &\qquad \text{VALUE=}\mathbf{TF}\llbracket I\rrbracket\ >\end{aligned}\tag{9}$$

$$\begin{aligned}&\mathbf{TF}\llbracket \text{“LINKS”}\ \text{“INT”}\ I\ \text{“=”}V\ \text{“SIZE”}\ N\ \text{“;”}\rrbracket \ \Rightarrow\\ &\quad <\text{INPUT TYPE=TEXT SIZE=}N\ \text{MAXLENGTH=}N\ \text{NAME=“}I\text{”}\\ &\qquad \text{VALUE=}\mathbf{TF}\llbracket V\rrbracket\ >\end{aligned}\tag{10}$$

The translation of identifier deserves a special mention. Since the actual data are not known until run-time, it is not possible to give a value to a field whether it is an input or an output field. The translation scheme has chosen to put special tags into the HTML pages which will be picked out and replaced before the page is to be sent to the user. The translation rule for identifiers is given below.

$$\mathbf{TF}\llbracket I\rrbracket \quad\Rightarrow\quad <\text{VICOMTE}>\ I\ </\text{VICOMTE}>\tag{11}$$

The scheme just presented is really a simplified version of what the system actually does. For example, there are various “hidden” fields and attribute values planted to facilitate processing on the server side, such as for the purpose of CGI programming [5]. Of course, there is also the validation code which is explained in the coming subsection.

6.3 Incorporating Validation Code

The generated HTML pages also incorporate validation code written in JavaScript. The choice of doing so is to allow constraint checking, both system or application constraints, to be performed on the client side so as to free the server from performing this job. More importantly, it eliminates the need of transferring data between the client and the server merely for the purpose of constraint checking. As a consequence, the user would find the system more responsive and more pleasant to use.

Validation of an individual field is performed immediately after data is input to the field. The user is informed of any error that is detected by the validation code. In the framework supported by JavaScript, a field is checked when it is changed, so an invalid input that is left unchanged will not be detected once more and so is true for a blank field that is never touched by the user. To cope with this shortcoming, all fields that are not blank are checked again when the user chooses to submit the form. Also, the post condition given in the form definition is checked at this very same time. With the post condition, blank fields that are not supposed to be blank are treated as errors.

Some of the translation rules presented in the previous subsection need to be changed to allow validation to be incorporated. Essentially, the “ONCHANGE” attribute has to be added to each input field. Taking rule 9 as an example, its modified version is given in rule 12. The “ONCHANGE” attribute is associated with a JavaScript function called “CheckInteger” (which will be examined later) to which the value of the of the data field (i.e. I.value) is passed as a parameter.

$$\begin{aligned}\mathbf{TF}\llbracket \text{“LINKS”}\ \text{“INT”}\ I\ \text{“SIZE”}\ N\ \text{“;”}\rrbracket \quad\Rightarrow\quad &<\text{INPUT TYPE=TEXT}\\ &\quad\text{SIZE=}N\\ &\quad\text{MAXLENGTH=}N\\ &\quad\text{NAME=“}I\text{”}\\ &\quad\text{VALUE=}\mathbf{TF}\llbracket I\rrbracket\\ &\quad\text{ONCHANGE=“CheckInteger(}I\text{.value)”}\ >\end{aligned}\tag{12}$$

The “CheckInteger” function is generated by the “JavaScript Generator” class. The function is planted between “<SCRIPT>” and “< /SCRIPT>” at the beginning of the generated HTML page to ensure that it gets loaded before the form is displayed (see Figure 9). The function makes use of the regular expression supported in JavaScript version 1.2. The regular expression of an integer value is defined as beginning (`^`) with an optional “+” or an optional “-” (`(+|-)?`) and ending (`$`) with one or more digits (`\d+`). Inside the function, the input “s” is tested against this regular expression. An invalid input will cause a dialog box to be displayed with the error message inside it.

Input fields of type “FLOAT” are checked in a similar fashion but using a different regular expression. Input fields of type “DATE” are treated as consisting of three component input fields of type “INT”. Dependencies between them, such as days in a month and leap years, are also checked. The validation of the post condition is performed when the user presses the “submit” button on the form which activates a planted JavaScript function. This function checks if

```
< HEAD >
...
 < SCRIPT LANGUAGE="JavaScript1.2" >

var IntegerRegularExpression=/^(+|-)?\d+$/

function CheckInteger(s) {
  if (IntegerRegularExpression.test(s)) {
    return true
  } else {
    alert("An integer is expected.")
    return false
  }
}
...
 < /SCRIPT >
< /HEAD >
```

Figure 9. Integer Checking using JavaScript

all non-blank fields are of the right types before it evaluates the post condition to decide if the form can indeed be submitted. The code for checking the post condition is obtained by translating the expression given in the "FINALLY" clause into an equivalent JavaScript expression.

6.4 Form Dispatching & Processing

A form is dispatched as a result of the decision made by the scheduler which determines which task is to be executed next, where and by whom. The request is then forwarded to the web server and this is why the first step of the form dispatching and processing cycle shown in Figure 10 begins at the web server. This activates the "HTML Page Generator" class to fetch the HTML template from the "File Manager" (step 2) and values of the parameters from the "Persistent Store Manager" (step 3). The "<VICOMTE>" tags in the template are then replaced by the current values of the parameters before the form is forwarded to the web server (step 4). The web server then takes over and patched the form with the necessary information so as to get the form delivered to the right place (step 5). When the form is filled in and passed the validation check performed by the embedded JavaScript code, it is submitted to the server (step 6). On receiving the HTML page, the server will locate the program in the server that is responsible for handling it. (The name and location of this program is defined at the time the system and the web server are installed.) Given a particular installation, the first translation rule given earlier says that the program named "**cgi-bin/vicomte**" found on the server "**www-rocq.inria.fr**" should be used. A "GET" value for the "METHOD" attribute in the "<FORM>" tag dictates that named parameters are to be passed as environment variables. With this information, the web server will launch the program which contains the "HTML Page Processor" class (step 7). This class will use the identifier of the form found in the hidden field "**Form_ID**" to determine the parameters used to generate the form and then the values found in the submitted form (now as environment variables) will be used to update the corresponding parameters (step 8).

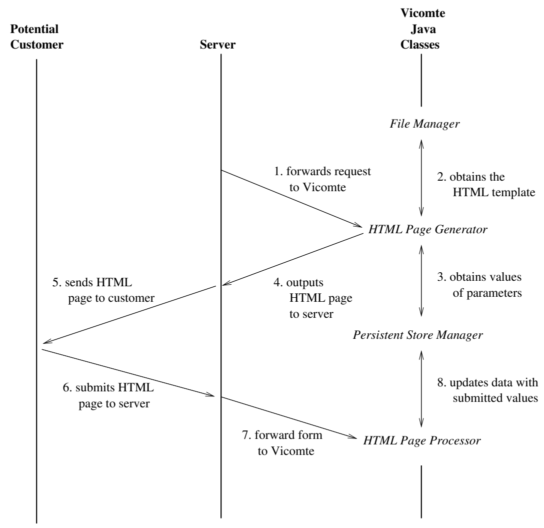

Figure 10. Form Processing using CGI

6.5 Storage Management

The "File Manager" class looks after the storage and retrieval of HTML page templates. It creates a directory hierarchy to reflect the structure of an application. The reason behind storing HTML page templates in the file system as opposed to the persistent store is because of the necessity to provide the user with the freedom to change the templates, for example to incorporate a company logo or introduce different colors for better visual effects. This freedom of editing costs an extra parsing when the form is to be dispatched as explained in the previous subsection.

The persistent store is used to manage both system data, such as form definitions, and application data, such as "c_name" mentioned earlier. The current implementation uses the persistence support provided by PJama [1, 17]. It has a great advantage that practically all kinds of data can be stored *directly* into the persistent store without going through any transformation as may be required in files and some databases. In other words, it provides orthogonal persistence to Java data in a seamless fashion. For instance, the abstract syntax tree representations of the forms are stored directly in the persistent store. Figure 11 shows how the abstract syntax tree ("ast") returned by parsing a *Valmont* specification ("parser.start()") can be stored in the persistent store under the name "Abstract Syntax Tree". This is all the program has to do – no flattening of the tree into records is necessary as in the case of a file. Upon termination of the program, the update to the persistent store is committed and "ast" is made persistent. The abstract syntax tree can then retrieved in its entirety from the persistent store using the name "Abstract Syntax Tree", for instance, for the purpose of generating a HTML page to be dispatched to the user. As a result of using a native Java persistent store, a lot of coding for input and output between modules has been saved. Moreover, PJama also provides transaction classes with which different transaction models can be constructed.

```
import org.opj.store.PJStore;
import org.opj.store.PJStoreImpl;
...
public static void main(String[] args) {
  ...
  AST ast = parser.start();

  try {
    PJStore ps = PJStoreImpl.getStore();
    ps.newPRoot("Abstract Syntax Tree", ast);
  } catch (PJSException e ) { ... }
  ...
}
```

Figure 11. Using PJama's Persistent Store

7 Conclusions

This paper presented a form management system which is a part of the *Vicomte* workflow management system. With the form management system, forms are defined in a form model using the *Valmont* specification language. The specification language allows one to define fields in the form, the type of data expected for the fields, various constraints over individual fields as well as between them. Some formating control is also possible. It is even possible to derive one form from another. The language is simple and concise but it does meet the need of many workflow applications. We believe it is suitable for many operational applications even though it would not be the best candidate for applications requiring sophisticated forms. Having said that absolute freedom to enhance the forms, e.g. using an HTML page editor, is accommodated which is not always possible with other systems that dynamically generate HTML pages. It is possible in *Vicomte* because the generated forms are stored in file and dynamic data are only included just before the forms are dispatched.

The implemented system turns form definitions into HTML pages with validation code embedded in them so that validation can be performed on the client side resulting in a more responsive and efficient system. The translation rules were presented in the paper. When a HTML page is to be dispatched, it is updated with the current values of the data found in the persistent store. A processor receives the pages submitted by the user and updates the persistent store with the new values.

Future work includes an assessment of the usability of the form management system, the development of a graphical or HTML-based user interface for form specification, inclusion of derived fields and possibly constraints over multiple fields, complex data types (e.g. arrays), and more format control similar to the "PICTURE" clause used in COBOL.

References

[1] M. Atkinson, L. Daynès, M. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.

[2] A. Barron and C. Lyskawa. *Microsoft Front Page 98*. Course Technology, 1998.

[3] D. Chan. A Document-driven Approach to Database Report Generation. In *Proceedings of the 9th International Workshop on Database and Expert Systems Applications (DEXA)*, pages 925–930. IEEE Computer Society Press, 1998.

[4] D. Chan and K. Leung. Valmont: a Language for Workflow Programming. In *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS)*, volume VII, pages 744–753, USA, Jan. 1998. IEEE Computer Society Press.

[5] S. Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly & Associates, Inc., 1996.

[6] N. Krishnakumar and A. Sheth. Managing Heterogeneous Multi-System Tasks to Support Enterprise-Wide Operations. *Distributed and Parallel Databases*, 3(2):155–186, 1995.

[7] P. Lawrence, editor. *Workflow Handbook 1997*. John Wiley & Sons, 1996.

[8] J. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates, Inc., 1992.

[9] Netscape Communications Corporation, U.S.A. *Netscape JavaScript*. http://home.netscape.com/comprod/products/navigator/.

[10] R. Purvy, J. Farrell, and P. Klose. The Design of Star's Records Processing: Data Processing for the Noncomputer Professional. *ACM Transactions on Office Information Systems*, 1(1):3–24, 1983.

[11] L. Rowe and K. Shoens. A Form Application Development System. In *Proceedings of the ACM International Conference on Management od Data (SIGMOD)*, pages 28–38. ACM Press, 1982.

[12] N. Shu, V. Lum, F. Tung, and C. Chang. Specification of Forms Processing and Business Procedures for Office Automation. *IEEE Transactions on Software Engineering*, 8(5):499–512, 1982.

[13] J. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[14] Sun Microsystems, Inc, U.S.A. *Java Compiler Compiler*. http://www.suntest.com/JavaCC/.

[15] Sun Microsystems, Inc., U.S.A. *Java Development Kit (version 1.1.6)*. http://java.sun.com/products/jdk/1.1/index.html.

[16] Sun Microsystems, Inc., U.S.A. *The Java Tutorial*. http://www.javasoft.com/docs/books/tutorial/index.html.

[17] Sun Mircrosystems, Inc., Sun. *PJama Release 0.4.6..12 (for JDK 1.1.6)*. http://www.sunlabs.com/research/forest/opj.main.html.

[18] D. Tsichritzis. Form Management. *Communications of the ACM*, 25(7):453–478, 1982.

[19] University of Georgia. *MTDes: METEOR Designer*, version 2.1 edition, Jan. 1997.

[20] S. Yao, A. Hevner, Z. Shi, and D. Luo. FORMANAGER: an Office Forms Management System. *ACM Transactions on Office Information Systems*, 2(3):235–262, 1984.