WebWork: METEOR$_2$'s Web-based Workflow Management System

John A. Miller, Devanand Palaniswami, Amit P. Sheth, Krys J. Kochut and Harvinder Singh
Large Scale Distributed Information Systems Lab (LSDIS)
Department of Computer Science
The University of Georgia
Athens, GA 30602-7404
email: *<jam,devanand,amit,kochut,hsingh>@cs.uga.edu*
URL: *http://LSDIS.cs.uga.edu*

ABSTRACT

METEOR$_2$ workflow management systems consist of both design/build-time and run-time/enactment components for implementing workflow applications. An enactment system provides the command, communication and control for the individual tasks in the workflow. Tasks are the run-time instances of intra- or inter-enterprise applications. We are implementing three versions of the METEOR$_2$ model: WebWork, OrbWork, and NeoWork. This paper discusses WebWork, an implementation relying solely on Web technology as the infrastructure for the enactment system. WebWork supports a distributed implementation with participation of multiple Web servers. It also supports automatic code generation of workflow applications from design specifications produced by a comprehensive graphical designer. WebWork has been developed as a complement of its more heavyweight counterparts (OrbWork and NeoWork), with the goal of providing ease of workflow application development, installation, use and maintenance. At the time of this writing, WebWork is being released to the LSDIS Lab's industrial partners.

1 Introduction

Workflow Management Systems (WfMSs) provide an automated framework for managing intra- and inter-enterprise business processes. According to the Workflow Management Coalition (WfMC), a Workflow Management System is a set of tools providing support for process definition, workflow enactment, and administration and monitoring of workflow processes [Hol94]. Application domains where workflow technology is currently in use includes healthcare, education, telecommunications, manufacturing, finance and banking and office automation. WfMSs are being used today to re-engineer, streamline, automate and track organizational processes involving humans and automated information systems [JAD+94, GHS95, Fis95, SKM+96, BSR96, SGJ+96]. The success of WfMSs has been driven by the need for businesses to stay technologically ahead of the ever-increasing competition in typically global markets.

METEOR_2 implementations consist of both design/build-time and run-time components. Run-time components are often referred to as *enactment systems*. This paper presents the WebWork Workflow Management System - which has been developed as a purely Web-based implementation of the METEOR_2 model. METEOR_2 [SKM+96] is an enhanced version of the METEOR model created by Krishnakumar and Sheth [KS95]. A prototype implementation of this model was developed by the same team [KS95]. Three prototype implementations of the METEOR_2 model are being developed - OrbWork, a fully distributed CORBA-based workflow enactment system [DKM+97], NeoWork, a CORBA-based workflow enactment system with centralized schedulers, and WebWork, a fully distributed workflow enactment system relying solely on Web technology. This paper focuses on WebWork which is the first implementation of METEOR_2 to be externally released. At the time of this writing, WebWork is being released to the Connecticut Healthcare Research and Education Foundation (CHREF), our partners in the HIIT/HITECC Advanced Technology Program funded by the National Institute of Standards and Technologies (NIST). A comprehensive real-world application for state-wide immunization tracking has been developed using WebWork, and is also being delivered. WebWork will soon be used at The Boeing Company, another of LSDIS's industrial partners.

An enactment system provides the command, communication and control (C^3) for individual application *tasks* participating in a workflow. Tasks are the run-time instances of intra- or inter-enterprise applications. Today they typically run independently or are tied together in ad-hoc ways. WfMSs tie these tasks together in a cohesive fashion.

The main components of a METEOR_2 enactment system are workflow schedulers, task managers and (application) tasks (see Figure 1). *Task managers* as the name suggests are used to control the execution of tasks (e.g., when they execute, where they get their input, what to do when they fail and where to send their output.). To establish global control as well as facilitate recovery and monitoring, the task managers communicate with *workflow schedulers*. It is possible for schedulers to be either centralized or distributed, or even some hybrid between the two [Wan95, MSKW96]. In fully distributed implementations, communication overhead is reduced if the scheduling functions are taken over by the task managers. Both OrbWork and WebWork follow this approach, while NeoWork has explicit schedulers.

Based on the characteristics of application tasks, an appropriate type of task manager is used. From a build-time point of view, task/task managers are organized in a hierarchical fashion with both atomic and compound tasks [SKM+96, DKM+97]. Typically, an enactment system will be based on a flattened version of a hierarchical design as is the case for WebWork. The structure

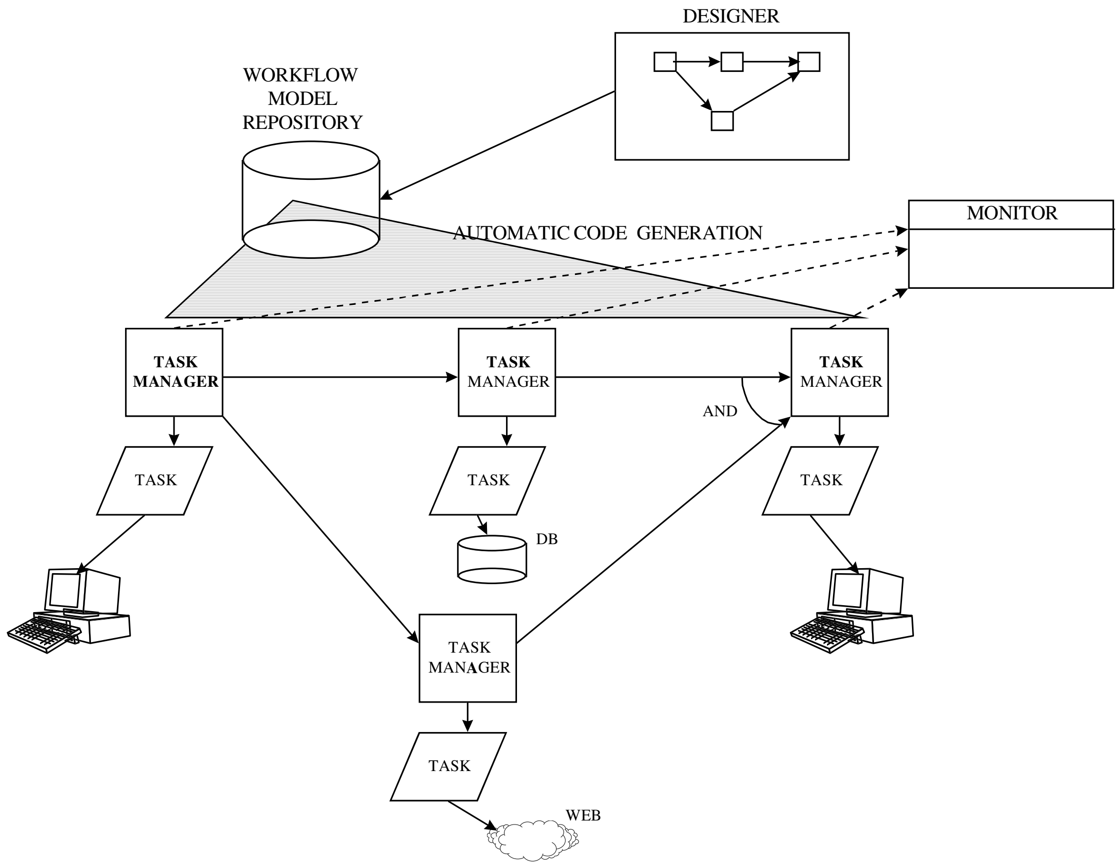

Figure 1: The METEOR$_2$ Model

and organization of task managers are based on their type. The METEOR$_2$ model prescribes the following task structures: NONTRANSACTIONAL, TRANSACTIONAL, WORKLIST, HUMAN_COMPUTER, WEB and TWOPC [Zhe97]. The concepts involved in using these task structures are adapted from [ASSR93, RS95]. This set of task structures models most types of activities one finds in modern information systems. It includes capabilities for manual (involving human interaction) and automated (without human interaction) tasks, transactional and non-transactional tasks as well as two-phase commit tasks. Presently, WebWork supports all of these except TWOPC.

The task structures are represented as directed graphs (acyclic as far as WebWork is concerned) [KS95, MSKW96]. The nodes in the graph correspond to the externally visible states, while the arcs correspond to permissible internal transitions. An internal transition is said to be controllable if it can be affected by another task via an inter-task dependency; otherwise, the internal transition is said to be uncontrollable. A NONTRANSACTIONAL task is used when an ordinary application that does not enforce atomicity or isolation is to be included in a workflow. Such a task can only be initiated or forced to fail (terminate early). The externally visible states of a NONTRANSACTIONAL task are *initial*, *execute*, *fail* and *done*. Optional dependency arcs may come into the *fail* state to force the task to fail.

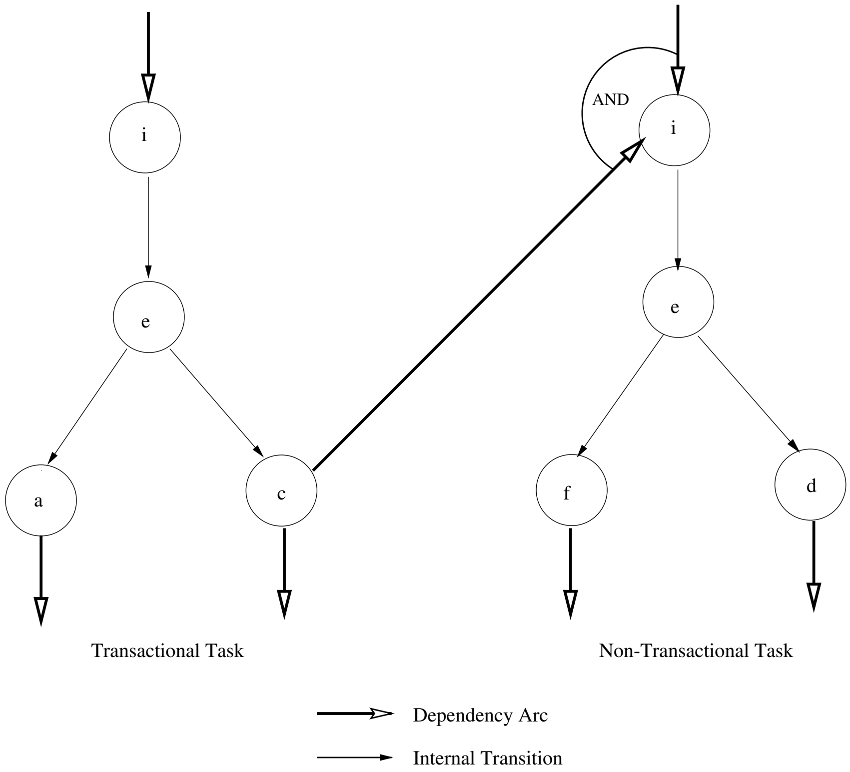

Figure 2: Task Structures

A TRANSACTIONAL task minimally supports the atomicity property and maximally supports all ACID (Atomicity, Consistency, Isolation, and Durability) properties [GR93]. The externally observable states of a transactional task are *initial*, *execute*, *abort* and *commit*. Optional dependency arcs may come into the *abort* state from outside the task to force the task to abort. Both of these types are considered to be automated tasks. A WORKLIST task is used to queue up work to feed automated tasks. A HUMAN_COMPUTER task is for purely manual tasks (e.g, a user fills in an HTML form). A WEB task may be interactive as well as invoke an application task.

The rest of this paper is divided into the following sections: Section 2 discusses the use of Web technology for workflow. It explains why Web technology is useful, briefly surveys some emerging Web-based workflow management systems, and highlights some of the advantages of WebWork. Sections 3 gives an overview of how workflow designs are mapped to run-time implementations. It also introduces an illustrative example workflow application called Immunization Tracking. Section 4 presents the components that make up the WebWork Enactment System in detail. Several design choices are discussed in this section (e.g., how to implement intertask dependencies, worklist mechanisms, automated tasks and recovery mechanisms). Section 5 highlights the workflow application development process. Considerable effort has been expended to make this process as rapid and straightforward as possible. Finally, Section 6 presents conclusions and discusses future work. A presentation at http://lsdis.cs.uga.edu/workflow gives additional details.

2 Use of Web Technology for Workflow

Two principal infrastructures - Web (World Wide Web) and CORBA (Common Object Request Broker Architecture) - are used to build METEOR$_2$ enactment systems. These infrastructures were carefully chosen for their functionality, availability, popularity and reasonable cost structures. Web technology is appropriate in two distinct and important ways.

First, the ubiquitous nature of Web browsers such as Netscape's Navigator makes it a natural user interface. Web browsers satisfy one of the primary concerns in application deployment - it allows users with any of the popular computing platforms to be able to participate in a workflow without any additional hardware.

A multitude of users (many of whom are not computer sophisticated) are already familiar with these browsers' easy-to-use interface. They presently see the interface as a way to access all sorts of information and perform simple tasks such as filling out forms. The unformity, wide availability and simplicity of the interface makes Web browsers an ideal user interface for workflow applications. This is particularly true in the healthcare domain where there is little time or inclination for special-purpose training.

Second, Web technology provides a solid communications infrastructure for building WfMSs. For WebWork, it is the only communications infrastructure, while for OrbWork it plays a supporting role to CORBA. We view CORBA as a key element for building very robust transactional workflow systems [SKM+96]. However, we believe it is unlikely that all organizations participating in a workflow (clinics and small hospitals, for example, in the healthcare domain) will want to purchase a CORBA product or, more importantly, will wish to maintain the software. It is more likely that they will have a Web server or at least access to a Web server. For this reason, METEOR$_2$ can support workflows in a Web-only (CORBA-free) mode. CGI programs/scripts are used to run tasks and coordinate the overall execution of workflows. The WebWork implementation relies on Web browsers, Web servers, HTML, JavaScript and CGI.

Many workflow management systems provide Web interfaces, but use other mechanisms for underlying communication/distribution (e.g., sockets, rpc or CORBA). Typically, these involve existing non-Web based engines for which Web interfaces are layered on top. In many cases, the workflow engine is centralized, with only access being distributed (multiple clients, single server). In the case of multiple servers, daemon processes must be installed and kept running at multiple sites for the system to work. We call workflow management systems that provide Web interfaces *Web-enabled*, while if Web technology is the only infrastructure used to build the workflow management system, providing both interfaces and communication/distribution, we say that it is *Web-based*.

A survey of existing commercial workflow systems that use Web technology [CS897], [SJ96], [PLS+96], [AAAM97] revealed that a majority of them are merely Web-enabled. But the trend appears to be to add more and more Web orientation to the WfMSs. This trend is illustrated by the following systems: ActionTech Metro, WebFlo, DartFlow and WebWork. Each one is successively more Web oriented, with WebWork being purely Web-based. A couple of other notable systems that utilize the Web and provide workflow capabilities are WebFlow [Cor97], PrISMS [NAS96] and Oz [KL96].

- ActionTech Metro [Tec97] by Action Technologies is a web-enabled workflow product, which supports many of the regular office automation processes. This includes passing of messages and automatic e-mails, keeping track of actions carried out by various users, and supporting the role paradigm. The Metro product is a Web interface to Action Technologies' conventional ActionWorkflow family of workflow products. Organizations such as Sandia National Laboratories are choosing products like ActionTech Metro, because its Web interfaces greatly reduce deployment costs.

- WebFlo [Con97] by Information Management Consultants consists of a useful product suite supporting Web interfaces to WfMSs (currently FileNet) as well as allowing the insertion of Java or CGI applications into work item forms to perform special functions. The product suite includes the following modules: WebFlo.Initiate creates work items to be processed through workflow. WebFlo.Status queries queues or tables to determine the status of work-in-process. WebFlo.Work allows users to participate in an existing workflow over the Internet. WebFlo.Docs provides document retrieval applications to view images stored in an organization's imaging system. WebFlo.AppBuilder is a windows-based wizard tool for creating custom applications. WebFlo.Secure is a windows-based tool for administering user privileges.

- DartFlow [CGN97] is a research prototype being developed at Dartmouth College. Their goal is to utilize Web technology to a greater extent than current WfMSs. They nicely summarize this in their paper [CGN97].

  "Unfortunately, WWW integration is mostly limited to offering a web browser interface to different proprietary workflow engines. In this paper we propose a different approach using open and portable

Web technology not only as a front end for the workflow client applications, but also for the implementation of the workflow enactment service and for administration and monitoring."

It uses Web-browser embedded Java applets as its front end and transportable agents as the back end. Java provides platform independence, and transportable agents are used to make DartFlow flexible and scalable. Once a user's password is verified the user is logged in and his/her worklist comes up. By clicking on the worklist, an HTML form is displayed. When it is submitted, a CGI script is invoked that creates a new agent to process the form. Since each transportable agent contains its own process description, DartFlow has the capability to adapt each instance of a task to its specific needs. The agent lives until it finishes the task for which it was created. DartFlow relies on Agent Tcl, also being developed at Dartmouth, in addition to Web technology to provide its foundation. It requires Agent Tcl servers to be running at participating sites and communication is provided by Agent Tcl using for example message passing or remote procedure calls (RPC).

We consider the fact that WebWork has been implemented as a functionally complete WfMS exclusively using Web technology as its communication infrastructure to be one of its major advantages. Test runs using WebWork have confirmed our original assumptions in choosing pure Web technology as the basis for WebWork. First, the WebWork WfMS was easy to install at all our test sites, with minimal changes to system configuration files. To begin developing workflow applications using WebWork, all that was required was for the associated Web server to be up and running. Second, the workflow application development process was very straightforward. After using the METEOR$_2$ graphical workflow designer to design the workflow application, the application developer needs to use the WebWork application builder which consists of a set of Web forms. This step typically requires very little user interaction to finalize/customize the specifications. Third, installing the application even across a distributed computing environment was straightforward and required human intervention only in terms of transferring tarred files and running installation scripts.

WebWork supports the development of workflow applications that can run in heterogeneous and distributed environments. Any number of organizations can be easily incorporated into a workflow - a Web server at an organization accesses its local database(s) using CGI programs, and these CGI programs can be accessed using Web browsers at various other organizations - thus facilitating a distributed client-server implementation. In addition, multiple Web servers are needed to integrate the various autonomous organizations and databases in distributed workflow applications. CGI programs can interact with existing heterogeneous DBMSs. Thus, already existing infrastructure can be efficiently utilized. Moreover, organizations participating in a workflow can easily use existing hardware, as the Web-based system is highly platform independent.

3 Mapping Design to Implementation

Tasks are designed using the METEOR Designer (MTD), a comprehensive graphical designer consisting of three integrated components - a map designer, a data designer and a task designer (see [DKM+97, Zhe97] for details).

As an example, let us consider the design of the initial segment of the Immunization Tracking workflow application (IZT) which has been developed for CHREF using WebWork. This application includes online interactions for the workflow application between CHREF (as the central location), healthcare providers (Hospitals, Clinics, Home Healthcare Providers) and user organizations (State Department of Health, Schools, Department of Social Services). The design consists of two compound tasks, AdmitPatient and TriageNurse (upper level map). Each of these compound tasks are expanded in complete subworkflow designs (bottom level maps). The tasks at this level will be implemented as part of the WebWork build process, which is the next step in the workflow application development process. The complete design will be integrated before the build process begins by using the flattening option provided by MTD. Screenshots of MTD designs for the IZT workflow application are shown in Figures 8, 9 and 10 in the Appendix.

After flattening the design, the individual tasks are meant to carry out specific application activities such as entering patient data or accessing databases. In the IZT workflow, tasks are responsible for providing access to patient data across multiple organizations (e.g., Hospitals, Clinics and CHREF). For instance, the Web server at CHREF provides data contained in the Master Patient Index (MPI), Master Encounter Index (MEI) and Immunization databases to various tasks. Tasks associated with an admit clerk (at a hospital or clinic) are responsible for retrieving patient-specific information (demographic data, encounter data, and immunization alerts) from the MPI, MEI, and Immunization databases. These tasks may be run on multiple Web servers. For efficiency reasons, it would be best to run the data entry tasks off the local Web server and tasks that access remote databases (e.g., the MPI at CHREF) off the Web server at CHREF.

The WebWork build process (described in greater detail later in this paper) uses specifications of the individual tasks to produce run-time code. Each task designed using the METEOR Designer is mapped into a cluster of WebWork processes. A design-level task is generally mapped to the following:

- a *Task Manager*,
- an *(Application) Task*, and
- a *Verifier*.

A task manager runs as a CGI program compiled from C++ source code which is fully generated from specifications. Its purpose is to read selected data from the previous task(s), prepare the data for the application task, invoke the application task, collect data from the application task, and prepare an output HTML page. The task manager is also responsible for handling and recovering from errors produced by the application task. An application task performs application specific operations. It need not even be aware that it is part of a workflow. Overall coordination and control are outside its domain of concern. Task applications may be coded in any language for which the organization has an available interpreter or compiler. A verifier consists of JavaScript code embedded in the HTML page produced by the task manager. Its purpose is to verify that the interaction with the user proceeded correctly, and then, based upon available data, select the next task(s) to execute.

A verifier consists of either explicit JavaScript code embedded in the HTML page produced by the task manager (in the case of manual tasks), or is integrated with the task manager code (in the case of automated tasks). For manual tasks, its purpose is to verify that the interaction with the user proceeded correctly, and then, based upon available data, select the next task(s) to execute. In the case of automated tasks, its purpose is to perform certain checks on the data and to select the next task(s) to execute.

These basic execution components are shown in relationship to each other in Figure 3.

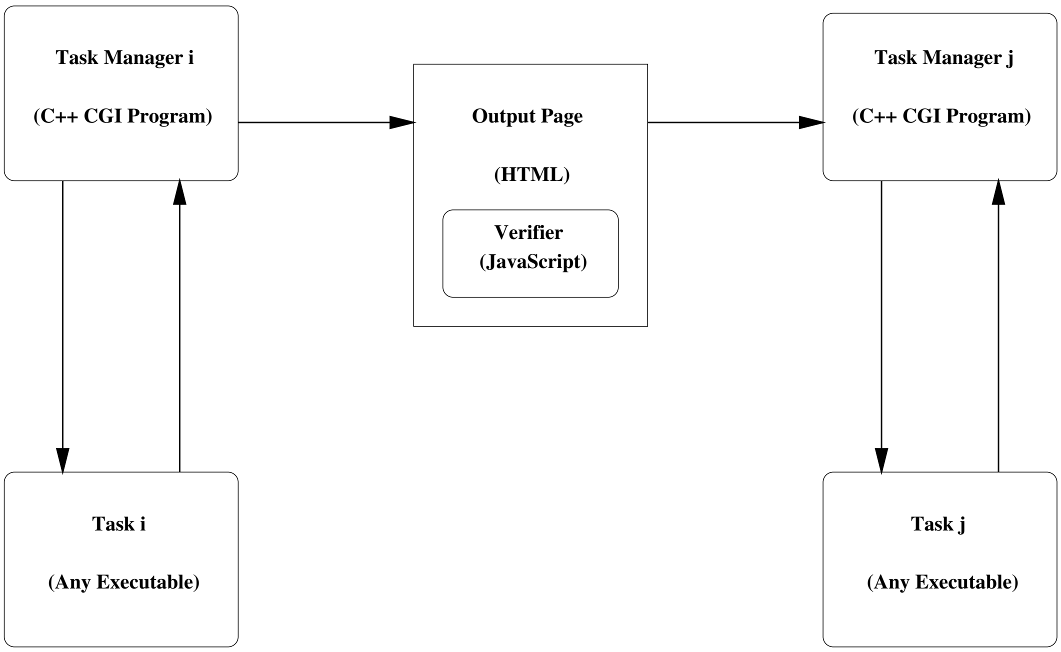

Figure 3: WebWork Execution Components

4 Components of WebWork

This section describes the components that make up the WebWork Enactment System. It discusses the functionality of tasks and task managers, the worklist mechanism, the differences between manual and automated tasks, inter-task dependencies, support for integrating Database Management Systems with WebWork, error handling and recovery mechanisms, and monitoring and tracking functions.

4.1 Task Managers

In WebWork, the responsibility for scheduling and coordinating the execution of tasks falls entirely with the task manager. Task managers are implemented as CGI programs making communication between task managers relatively easy. For manual tasks, communication is from CGI program to Web server to Web browser to Web server to the next CGI program, while for automated tasks, the browser is removed from the loop (see Figures 5 and 6 in Section 4.6).

4.1.1 Task Manager Organization

A task manager CGI program follows a very regular structure, so much so that the main function is generic for all task managers. This is the result of a deliberate design to make automatic code generation and code reuse easier.

```
int main ()
{
    uncgi ();
    Data_Tuple data (NO_DATA_ELEMENTS);
    if (tm.initialize (data)) {
        if (tm.execute (data)) {
            tm.output (data, DONE);
            exit (0);             // success
        } else {
            tm.output (data, FAIL);
            exit (-1);            // execute failed
        }; // if
    } else {
        tm.output (data, FAIL);
        exit (-2);                // initialize failed
    }; // if
}; // main
```

Coding is simplified by using the uncgi call which is part of the Un-CGI library (Grimm, 1997). To link his code with our C++ code we converted his source to ANSI Standard C. This library makes it very simple to process CGI input. Calling uncgi at the beginning of the program places all the input values into environment variables. Subsequently, these values can be easily obtained programmatically by making a call such as:

```
Patient_Name = getenv (WWW_Patient_Name);
```

where Patient_Name is an input to the task manager (CGI program).

Next, the initialize method reads data from the previous task(s). If no problems occurred, it returns true (1) so that execution can begin. The execute method calls the do_task function to carry out the application task. do_task is a wrapper that runs the actual application task. do_task may throw a variety of exceptions, which are caught by the execute method. The execute method will call do_task up to MAX_TRIES times. The default is two tries.

The behavior of this generic task manager is customized in two ways. The principal customization is based on the task type which is used to select the appropriate task manager class (see below). A secondary customization allows variations in the data, conditions and outputs. This information in provided in an included (.h) file in the form of initializations of various aggregate variables. This approach makes code generation easier. The include file changes for each task, while the implementation file (.cc) remains the same.

4.1.2 Task Manager Classes

The main program for a task manager simply sequences through the initialize, execute and output methods of a task manager class. For each supported task type, there is a task manager class. The present release of WebWork supports NONTRANSACTIONAL, TRANSACTIONAL, WORKLIST, HUMAN_COMPUTER and WEB task types. The first three are automated tasks, while the last two are manual. Classes for each of these types are derived from the following abstract base class.

```
class TaskManager {
public:
     TaskManager (char* name);
     virtual int initialize (Data_Tuple data);
     virtual int execute (Data_Tuple data);
     virtual int output (Data_Tuple data, FinalState fs) = 0;
     virtual ~TaskManager ();
 private:
     char* task_name;
 }; // TaskManager
```

It is straightforward to derive other classes from this base class or any other task manager class to support new task types.

4.2 Tasks

Application tasks (or simply tasks) may be coded by a workflow developer, may already exist in the WebWork *task library*, or may be an existing external (or even legacy) application. As a general rule, we envision the applications tasks to be workflow unaware. They should be able to run individually as ordinary applications. This rule seems obvious when one thinks about integrating existing applications, but task coders might be tempted to violate it for short term coding expediency. For now, we have decided not to enforce this rule, but recommend that it be followed so that testing/debugging and maintenance become easier.

The way in which task applications are run is an important design issue. In WebWork, a task can be run by a task manager in one of two ways.

1. **Task Function**. An application task can be run as a function called by the task manager and executed within the task manager's process. The do_task function within the task manager will simply copy appropriate data elements into the *in* and *inout* parameters of the task function, call the task function, and then, copy *out* and *inout* parameters into appropriate data elements.

2. **Task Process**. An application task can also be run as a subprocess of the task manager. The do_task function within the task manager will marshal arguments for the task process, run the task process passing in the arguments, and then, read the outputs produced by the task process. The exact mechanisms are operating system dependent, so we developed our own functions (e.g., invoke) to make the necessary systems calls (e.g., popen, pclose, pipe, fork, exec). These functions use conditional compilation based on the value of the PLATFORM symbol (UNIX or NT). One simple implementation of invoke that works on UNIX systems is to use the following system call.

   ```
   child_output = popen ("app_task arg1 arg2 ... argn", "r");
   ```

   This runs app_task as a child process and creates a pipe (interprocess communication) to transfer the child's standard output to the parent's standard input. The child process (application task) receives input from the task manager (parent process) from command-line arguments, and sends output to the task manager by writing to the pipe.

The choice of which option to use, a task function or a task process, is up to the workflow designer/developer. If the application task is simple enough it may be coded as a function, and if it is general enough, it may already be in the task library. On the other hand, one may wish to invoke an existing program or write a new task as a program. This increases the flexibility of task coding. Such tasks can be written in any language, be compiled or interpreted, and may be executed independently of the workflow. Performance and reliability can be used as additional issues to contrast the two approaches. Generally, running tasks as functions should be faster (no fork and exec overhead), while running them as processes should be more reliable. In both cases, signal handlers can be used to catch fatal errors (e.g., Segmentation Faults), but when an application task runs within the task manager's memory, severe damage to the task manager may have already been done before the signal is sent. This type of protection is important since task code is typically coded by the workflow application developers and as such is less likely to be rigorously tested.

4.3 Verifier

Rather than have a centralized scheduler, local scheduling or control decisions are made for each task. For manual tasks, the verifier executes on the browser side and consists of *JavaScript if statements* which are automatically generated at run-time based on the dependency conditions specified in the workflow design. Once a user is finished with his/her interaction with the page, s/he clicks its submit button. This will invoke the JavaScript code which will select the next task(s) to be executed. Using all the data available from the output HTML page and any user input, the verifier evaluates the dependency conditions and selects the next task manager(s) to invoke. The task manager incorporates the JavaScript as part of its output HTML page. For automated tasks, the verifier executes on the server side and is embedded within the task manager. It is also coded in C++ rather than JavaScript.

An important function performed by the verifier is to check the values in data entry fields to ensure that they do not violate constraints. Typically, some of the data elements (or attributes) for a task will correspond to data entry fields in an HTML form. The verifier checks this data right at the browser where it should be checked. If the values violate the constraints, the user must re-enter the values. Constraints on data attributes or elements are gleaned from the designer specifications, or optionally, specified by the workflow developer at build time (see [Pal97] for details). The verifier code that performs these constraint checks is generated automatically at run-time based on these specifications.

4.4 Inter-Task Dependencies

A collection of tasks are harnessed into a productive workflow by specifying and implementing inter-task dependencies to manage the flow of data and control between the tasks. The METEOR$_2$ model is flexible enough to allow this to be done in a variety of ways (centralized vs. distributed scheduling, tight vs. loose coupling of data and control dependencies) and using entirely different distributed processing infrastructures. WebWork, OrbWork and NeoWork exercise most of these options, so in the near future we hope to gain some useful insights with respect to the various alternatives (see [MSKW96]).

An inter-task dependency specifies the conditions under which one task can enable the execution of another task. [1] We use the phrase "enable the execution" since a task may require more than one predecessor task to finish before it is fully enabled, as illustrated in Figure 2.

An inter-task dependency consists of the following three elements:

- A *Dependency Arc.* This is simply an arc from one task-state pair to another task-state pair (e.g., done@IndentifyPatient to initial@CheckAlerts).
- A *Dependency Condition.* This is a Boolean condition based on available data which is used to determine if control should flow through the arc (i.e., whether the next task should be enabled).
- A role in the next task's *Activation Precondition.* A task may have several dependency arcs incident upon it. If the arcs are OR'ed together, any one of them may activate the task, while if they are AND'ed together, the task must be enabled by all predecessors. In general, any combination of AND's and OR's is allowed and should be given in a disjunctive normal form (OR of AND's).

The WebWork implementation conforms closely with the METEOR$_2$ model. It uses distributed scheduling and tightly couples the data and control dependencies. All scheduling decisions are made by the task managers/verifiers running on various Web servers/browsers. Data and control information is communicated between the tasks through HTML pages in the case of manual tasks, and directly from one task manager to the next in the case of automated tasks (this mechanism is described in detail later). When the control flows, the data flows along with it. For a manual task, the HTML page contains a form for a user to input data or possibly confirm that the user has finished interacting with the page. The output (standard output) of a task/task manager (CGI program) is written as an HTML page. This page contains the JavaScript code which makes the flow of control decisions.

WebWork allows a dependency arc to be drawn between any task's final states (e.g., done and fail) and another task's initial state. It allows any Boolean condition involving the source task's own data elements to be specified as a dependency condition. This permits alternative paths to be taken (e.g., in the Admit clerk subworkflow, upon finishing with the CheckAlerts screen, the user may

[1] Note that NeoWork also allows an inter-task dependency to influence the execution of a task that has already started, e.g., a TWOPC task.

select any of three next tasks; see Figure 9 in the Appendix). Note that in other implementations such as a centralized one, it is possible to base the dependency condition on more than just the attributes known to the source task.

Input data received by a task is stored either in the task's worklist or log (see below). A worklist (or log) entry is not considered complete until all prerequisite information is added (e.g., if two incoming dependency arcs are ANDed together, then the task must receive input from both prerequisite tasks before it can perform its function). More details on how activation preconditions are implemented can be found in the next subsection.

Let us now briefly trace the flow of control and data from one task to the next, specifically from IndentifyPatient to CheckAlerts. During the first task, IndentifyPatient, an HTML form will be displayed. The users will fill in the form and submit it, at which point the JavaScript verifier will take over and enable the next task, in this case CheckAlerts. This enable will activate CheckAlerts which will read the name-value pairs produced from IndentifyPatient's form and continue its execution. If the next task has a worklist, a more complex mechanism is necessary as explained in the next subsection. If the next task could be performed by someone else (e.g., another admit clerk or a nurse), then the next task must have a worklist.

4.5 Worklist Mechanism

The worklist mechanism provides another type of communication between tasks. A worklist can be used to maintain a list or a queue of tasks that are ready to be performed when the entity performing it is ready or available. Any manual task may have a worklist. Although less commonly used, an automated task can also have a worklist to support the queueing of objects.

A worklist is essential for coordinating work between two or more workflow users. A good example is an admit clerk sending patients to a nurse. This is achieved by the final task of the first user (an admit clerk) simply writing a work entry into the nurse worklist. Using the client-pull method, the first task for the next user (a nurse) periodically polls the worklist rewriting the worklist frame (see Figure 13) based on the currently available work entries.

The actual implementation of this mechanism within WebWork is somewhat involved. Imagine again an admit clerk and a nurse, with the admit clerk using browser1 and server1, and the nurse using browser2 and server2 (see Figure 4). The clerk's last task (CGI program) outputs an HTML page which goes to server1 and then to browser1. This will cause another CGI program to be run by server2. This CGI program will write the data (the clerk's last task's data) to the nurse's worklist (stored in a file or database). A third CGI program will periodically read the worklist and redisplay the information in the worklist frame. Finally, when a entry in the displayed worklist is selected, a fourth CGI program will start or invoke the application task.

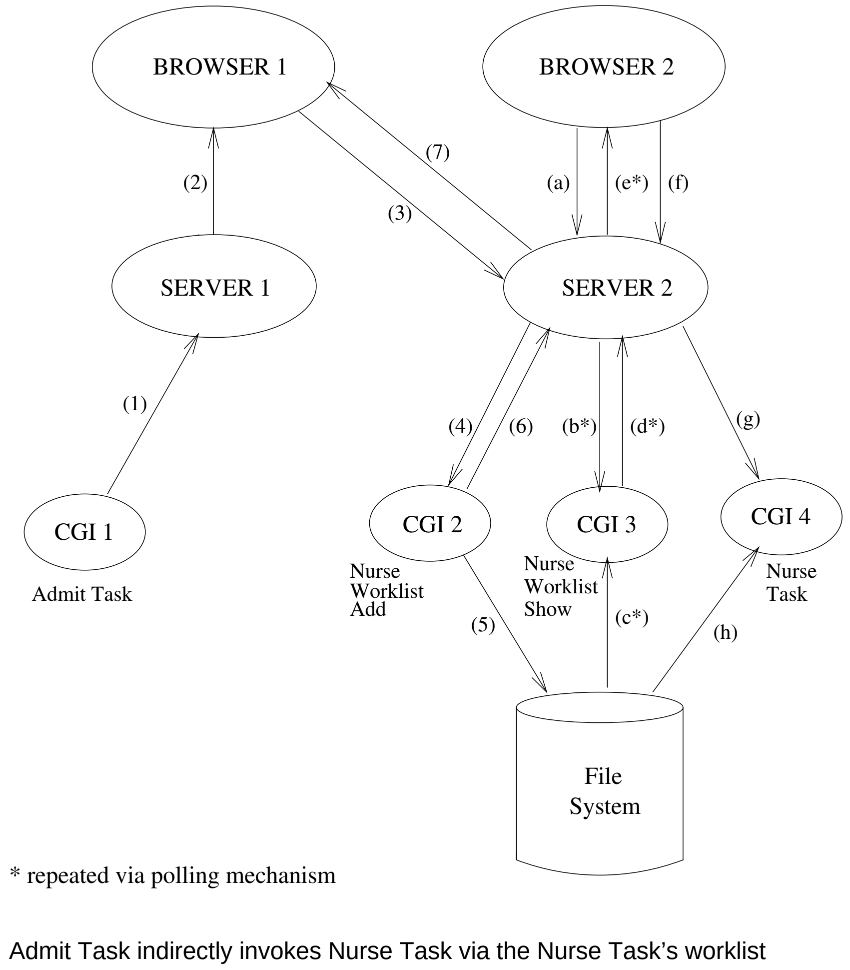

Figure 4: Worklist Mechanism

At the design level, two tasks are involved. At the implementation level, the clerk's task maps to one CGI program, while the nurse's task maps to three CGI programs (one to "add" entries to the worklist, one to "display" the worklist, and one to "perform" the application task).

A particularly convenient aspect of this mechanism is that the worklist is shared by all nurses. It belongs to a particular task type (e.g., ShowPatientData). More than one nurse be may perform this task, so that at any particular time, this one task type may have multiple task instances. Each task instance would require their own "display" and "perform" CGI programs. However, one invocation of the "add" CGI program updates the worklist for the entire group.

In the automated case, a WORKLIST task is implemented as two CGI programs ("add" and "perform"). The "display" step is not necessary, since the worklist is not visible to users. The "perform" CGI program will be run periodically and if entries exist on its worklist, it will pull them off and begin working on them.

For tasks that do not have a worklist, an input log is used instead. Such tasks are implemented as a single CGI program which combines the functionality of both the "add" and "perform" steps. The "add" step will append the input data to the log and then evaluate the activation precondition. If it evaluates to true, the perform step will be carried out; otherwise the program will exit.

4.6 Implementation of Automated Tasks

Although the Web is ideally suited for interactive workflows consisting of manual tasks, WebWork supports automated tasks as well as fully automated workflows.

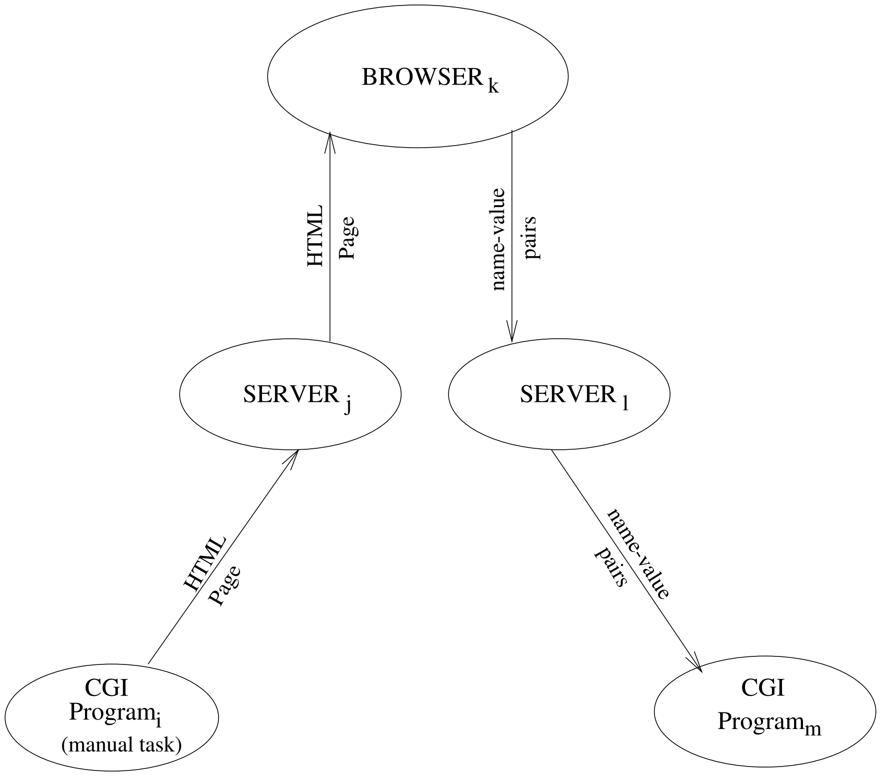

Figure 5: Invocation of Manual Tasks

Ordinarily, communication between CGI programs follows the route shown in Figure 5, that is, from CGI i to Web server j to Web browser k to Web server l to CGI m, where j may or not be equal to l. For automated tasks, we simply bypass the browser by tricking the Web server as shown in Figure 6. For automated tasks, CGI i establishes a socket connection with a Web server l and sends a message structured like a message from a browser. The Web server will respond to this message by invoking CGI m. CGI m need not be aware of whether it was invoked by a browser or by another CGI program. A restriction, however, is placed on tasks which follow an automated task; such a task must itself be an automated task or have a worklist. The reason for this is simple. When a workflow initiates an automated task, it is useful to allow the user to continue interacting

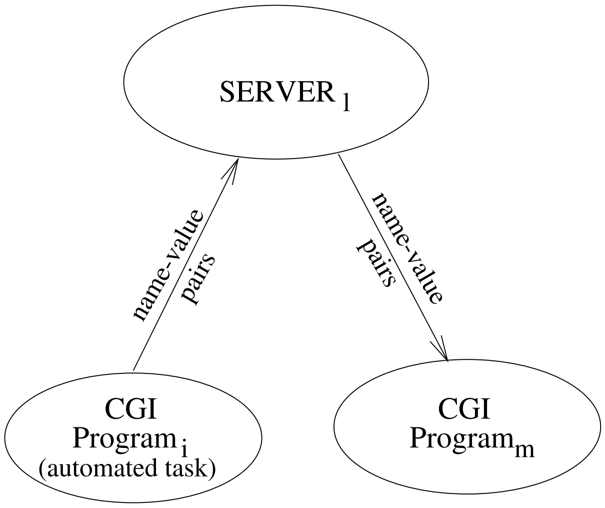

Figure 6: Invocation of Automated Tasks

by moving on to another manual task. When the automated task finishes, it does not make sense to interrupt the user by writing the results of the automated task to his/her browser's screen. Rather, the results should be placed on a worklist so the user may select them when ready.

4.7 Database Access

As with any advanced client-server software, flexible and convenient access to a variety of DataBase Management Systems (DBMSs) is of utmost importance. WebWork supports access to the two most important database technologies of today, *relational* and *object-relational*. For maximum flexibility, access is provided in three distinct ways, (1) *embedded SQL*, (2) *ODBC API calls* and (3) *native DBMS calls*. SQL-2 is the standard query language for today's relational DBMSs (e.g., Oracle 7.0) and forms the core of today's object-relational DBMSs (e.g., Illustra, Informix Universal Server). ODBC (or Object DataBase Connectivity) is a standard low-level Application Programming Interface (API) and is available with many of today's popular DBMSs. As a last resort, native calls such as Oracle's OCI or Illustra's MI may be used. The WebWork task library will contain functions to access (query and update) Oracle, Illustra, Informix, Msql and MS-SQL server. Because of the generality of embedded SQL and especially ODBC, many of these function will work with a variety of DBMSs. Furthermore, a developer can custom code his/her own database access task and, when it is sufficiently tested to the satisfaction of the workflow administrator, add it to the task library.

Database access is convenient in the sense that the functions we provide are very easy for the developer to use [Pal97]. As part of the design/build process, a workflow developer can specify a SQL command, such as the one shown below.

```
select LastName, FirstName
from PatientData
where PatientID = <%PatientID%>
```

At build time, WebWork will replace all special tags such as <%PatientID%> with accesses into the Data Tuple, e.g., at the PatientID index position. Then at run time the appropriate value will be used. The do_task wrapper will prepare the SQL command, and then execute the query or update command. In the case of a query, results will be obtained from the database and placed in the Data Tuple. Since the do_task wrappers are fully generated, no manual coding of application tasks is needed for standard types of database access.

4.8 Error Handling and Recovery

A workflow represents a very complex computational activity. There are many tasks whose execution needs to be coordinated. Some of these tasks may be newly developed and unfortunately not tested as thoroughly as they should be. Also, resources may be temporarily unavailable and the workflows must adapt to this. Therefore, comprehensive error handling and recover mechanisms should be included with any industrial strength WfMS.

Any WfMS must handle common errors and failures of tasks if it is to be of practical use. In addition, error handling and recovery at the task manager and workflow levels are also important [WS97]. WebWork's goal is to provide useful and practical error handling and recovery with minimal overhead. Presently, the error handling and recovery mechanisms simply rely on three types of persistent data: (1) data maintained by the Web servers, (2) data in the WebWork worklists or input logs (one for each task), and (3) data recorded in WW_error_logs (one for each Web server).

For WebWork, we have classified ten types of errors or failures [2] that may be encountered.

- No. 10: Data Entry Errors on Forms. These are handled by the verifier. If the values entered violate the constraints, the page will be redisplayed with an indication of the error. Data entry fields obtain constraints in one of two ways. First, the data types specified for the attributes by the graphical workflow designer can be used to constrain the data entered. For example, if the attribute is of type int, then only digits should be entered. In addition to constraints based on data types, two other types of constraints may be specified, namely, Not-Null and Bounded. Not-Null forces the user to type something in the data entry field for the attribute, while Bounded allows lower and upper bounds to be placed on the value entered. Examples

[2] This classification, informally known as Dev's Top Ten List, presents errors/failures in approximately increasing order of severity.

of bounds for int, float and string are given below. As mentioned previously, the verifier code to enforce these constraints is automatically generated when the task executes.

```
Age            [18, 65]
Price          [9.99. 19.95]
Group_A_Member ["A", "M"]
```

- No. 9: Task Error. In this case, a task returns an error code. The do_task wrapper receives this error code and maps it to an exception. The execute method within the task manager will attempt to handle the exception and then reinvoke do_task. The mapping between error codes and exceptions is purposefully set up to be very flexible. The mapping may be coarse (the default) or fine. A coarse mapping is set up by simply defining a symbol.

  ```
  #define ERROR_CODE_MAP "Enhanced"
  ```

  Enhanced is the default value. Other possible values for this symbol are given in the table below. Note that when the ERROR_CODE_MAP is set to Customized, a mapping table must be entered and stored.

Map Type	Success	Warning	Error	Ranges
Enhanced	$== 0$	> 0	< 0	–
Standard	$== 0$	–	$!= 0$	–
Inverted	$!= 0$	–	$== 0$	–
Customized	$== 0$	$> k1$	$< k2$	$[k3, k4], [k5, k6]$

Table 1: Types of Error Code Maps

- No. 8: Task Failure. A task failure occurs when a task dies in the middle of its execution (e.g., because of a segmentation fault). If the task is a task function running within the address space of the task manager process, this type of failure is potentially dangerous. We deal with this by having the task manager set up signal handlers to prevent the task manager from crashing. The handler writes the relevant information about the failure to WW_error_log and then calls the output fail method of the task manager. There is no guarantee that this will work, since the faulty task function may have wreaked havoc with the task manager's memory. A more reliable way to run an application task is as a task process. In this case, it has its own address space so that the task manager's memory will be spared any deleterious effects. The task manager detects that the task (child process) has died when its read operation on the pipe returns with an error code. At this point, the do_task function raises a TaskFailureException. Typically, the execute method will handle this exception by restoring and checking the data tuple, and re-executing do_task up to MAX_TRIES times. Since the previous call may have corrupted the data tuple, it is restored from a copy.

- No. 7: Repeated Task Errors or Failures. Except for the case of task failure within the address space of the task manager, typically the manager will attempt to run the task again. (We do not re-execute task functions by default since we felt that it would be too risky to run the task again.) On second and subsequent retries, the task manager sleeps for increasingly longer periods of time in the hope that possible resource problems may clear up over time. After a task manager has retried a sufficient number of times, it gives up and and records the errors in the WW_error_log. Then it transitions to the fail state which enables the next appropriate task. It is also up to the output fail method to provide an appropriate display. Successful handling of the problem is now dependent on the overall design of the workflow application. If appropriate alternative paths are designed in, then the workflow should be able to adapt to such failures.

- No. 6: Task Manager Error. Even though task managers can be thoroughly tested, Murphy's Law tells us that errors will occur within task managers. The most likely errors are those involving system calls (e.g., opening files, pipes or sockets). All such suspect operations are guarded by if statements. If an error occurs, the task manager will write a message to the WW_error_log and then call the output fail method.

- No. 5: Task Manager Failure. In this case, a task manager dies in the middle of its execution. This can happen because of internal (e.g., an unhandled signal due to an internal error) or external reasons (e.g., someone with root access kills your process). If the machine running a task manager fails, clearly the task manager will fail as well, but we consider this to be a more severe failure since task managers run on machines that host Web servers. If a task manager for a manual task fails, the event will be noticed by the user since an error message will be displayed on his/her browser. If the user simply clicks reload, the Web server will run the task manager (CGI program) again. Since all inputs to the task manager have been saved in either a worklist or an input log, no information will be lost. If the task manager fails repeatedly, the user should report this to the workflow administrator. If a task manager for an automated task fails, a user may become aware of this at some later time, at which point, they should inform the workflow administrator. Some CGI program failures may be recorded in the Web server's error_log, so the workflow administrator should periodically check the log.

- No. 4: Enable Failure. This happens when one task manager is unable to invoke another task manager. If this happens, re-attempt to start the task manager from the display page of the previous task manager, making sure that all data entry fields contain correct data. If the error persists, the user should inform the workflow administrator. A couple of likely reasons for this is that the CGI program executable file is not available or it is not executable. If the executable file is not in the appropriate directory, the administrator can reinstall it. If it is not executable, the administrator can change its permissions. For automated tasks, such failures are likely to show up as errors on socket operations or Web server errors, so the appropriate error logs should be examined.

- No. 3: Lost or Corrupted Input Data. It is possible that somewhere in the transmission or processing of data it gets corrupted. The TCP/IP protocol will perform such error checking for end-to-end transport. Errors will cause retransmission. At present, WebWork assumes that existing mechanisms such as those provided by TCP/IP will be sufficient. In a future version, we may add some sort of checksum mechanism of our own that computes a result based on the transmitted data. If the receiving CGI program finds a checksum error, it could use the "location" directive to redisplay the previous page. This would require output pages to be stored in files.

- No. 2: Web Browser Failure. If a user's Web browser (e.g., Netscape Navigator) crashes, the user should simply restart the browser and run a role-initiating CGI program for one of the roles (e.g., NurseRole) s/he is authorized to perform. This program will authenticate the user and then initiate his/her interface into the workflow. This interface is displayed as a top header frame and has hyperlinks to the first task for his/her role as well as all tasks that have worklists. The user can then return to the worklist s/he was working on before the browser failed. Automated tasks should not be hindered by a browser failure, so we need not worry about them.

- No. 1: Web Server Failure. Web servers are rapidly becoming lifelines for organizations. Web server downtime will be problematic to WebWork as well as the organization as a whole. Consequently, Web servers should be run on reliable machines and typically dedicated to the role of providing Web service. Therefore, Web server failure should be rare. If it does happen, it is the responsibility of the Web administrator to restart the Web server. In conjunction, the WebWork administrator (may or may not be the same person as Web administrator) should restart WebWork (see WebWork Administrator's Guide for details [Pal97]).

More sophisticated recovery capabilities will be provided with OrbWork [WS97] and some of them have analogs for WebWork which may be incorporated in a later version of WebWork. We first wish to field test WebWork to obtain feedback on how adequate this low overhead approach to recovery is.

4.9 Monitoring and Tracking

Monitoring and tracking of workflow instances is important for administration, management and recovery of workflows. Monitoring is used to determine what is happening at the current time, while tracking allows one to investigate what has happened. Both of these capabilities are supported in WebWork with tools that access logs/worklists created by WebWork as well as those created by the participating Web servers. WebWork has a simple forms-based interface that allows administrators to examine the logs and worklists in order to determine the status of workflow instances. For example, a user may call up the WebWork administrator asking why nothing has appeared on his/her worklist. The administrator should be able to diagnose the problem quickly by using the monitoring tool. A major failure may have occurred or maybe the user upstream in the workflow may not have completed his task(s). Later versions of WebWork may include a more sophisticated monitoring tool that displays status information in graphical form (similar in appearance to the graphical designer), or even shows an animation of active workflows.

5 Workflow Application Development

WebWork was designed to make workflow application development as straightforward and rapid as possible. Rapid workflow application development is accomplished by using a powerful, yet user friendly graphical Workflow Designer, by keeping the rest of the build steps quick and simple, and by providing an extensible task library for functions and programs.

By following a simple three step process, a workflow application developer is able to create a prototype workflow application. These steps use (1) the Workflow Designer, (2) the Workflow Builder and (3) the Workflow Generator. At this point, a complete runnable prototype is available for testing. This prototype uses generated default screens (HTML pages) and stubs for application tasks. Although not usable at the production level, such prototypes are very useful for refining and testing the workflow design. Once the design has been refined through an iterative process, the developer is ready to finalize/customize the workflow application. This is done by carrying out the final two steps in the development process: (4) recoding of application tasks and (5) customization of HTML pages. Each of these five steps is depicted in Figure 7 and explained in greater detail in subsequent subsections.

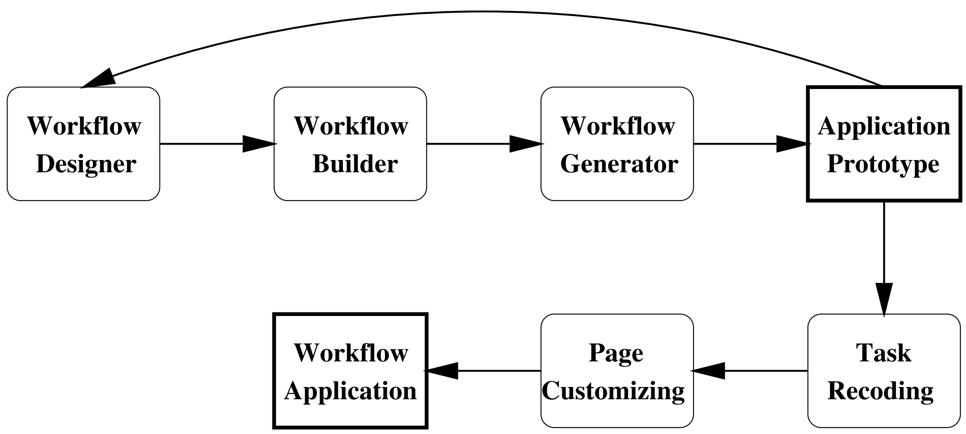

Figure 7: Rapid Workflow Application Development

5.1 Workflow Designer

The developer first makes a directory off of $WW_HOME/apps in which to create a workflow application, call it mywf. S/he then changes to this directory and invokes the designer, MTD. This can be done with the following commands.

```
> cd $WW_HOME/apps
> mkdir mywf
> cd mywf
> mtd &
```

The MTD workflow designer consists of three graphical designers, a map designer, a data designer and task designer (see Figures 8, 9, 10 for examples). Each of these integrated designers must be used to produce a complete design. See the MTD Workflow Designer's Guide for more details [Zhe97]. The end result of a careful and complete workflow design will be a .wil (Workflow Intermediate Language) file. This should contain adequate information for generating the code for a workflow application.

5.2 WebWork Builder

After the design is completed, the developer is ready to use the Workflow Builder. This is done from a Web browser by opening the URL for the webwork_build CGI program (see Figures 11, 12 for sample screenshots). This is a relatively quick step that allows the developer to review and check the default specifications for each task. If everything is acceptable, the developer simply clicks submit; otherwise, they may carefully change some of the default settings (i.e., customize the specifications a bit).

5.3 Workflow Generator

At this point, most of the work required to produce a prototype workflow application has been done. Only the workflow generation step remains. Entering the following commands from the main development window will cause all the executable CGI programs and associated HTML files to be created.

```
> make
> make source
> make compile
```

In particular, typing "make" (or "make makefile") will cause a detailed makefile to be generated. Typing "make source" will cause source code to be generated for all the task managers. Some code generation also occurs at the application task level. In particular, if an application task is implemented as a function and that function is not found in the task library, a header (.h) file as well as a stub (.cc) for it will be automatically generated in the current directory. The stubs will pass back fixed values for parameters, allowing for rapid prototyping of a workflow application. Once the code generation is complete, the source code can then be compiled and linked to form executables by typing "make compile".

5.4 Manual Task Recoding

If all the application tasks were not found in the task library (tasklib) or in the path specified by the designer, a relatively small amount of manual coding is required. In the case of task functions, the stubs need to be recoded, while in the case of task processes/programs, they will need to be fully coded from scratch. One can easily switch from task function to task processes simply by toggling a radio button on a Workflow Builder screen.

5.5 Page Customization

The Workflow Generator will produce an HTML file appropriate to a particular task. This HTML file determines how output pages of CGI programs will look. It will have special tags within it, for which an executing task/task manager CGI program will substitute actual values (e.g., values retrieved from a database). As a simple example, consider the fragment of HTML code given below.

```
<input type = text name = "LastName" value = "<%LastName%>">
```

During run time, before the page is output, the value stored in the Data Tuple at index position LastName will be substituted for <%LastName%>.

The default generated HTML page may be sufficient. If it is not, it can easily be customized by the developer using an HTML editor of his/her choice.

5.6 Application Installation

A developer may install either a prototype or a full implementation of a workflow application. In either case, the application will simply consist of a set of CGI program executables and associated HTML files. Manual tasks are CGI programs that produce HTML/JavaScript pages to be displayed on browsers (Figure 13 shows a screenshot of such a running task), while automated tasks are CGI programs that runs independently of any browser.

For installs on the same network file system, the developer simply types the following.

```
> make install
```

The "make install" command will install each executable CGI program in the appropriate directory on the appropriate Web server. Let us suppose that for the IZT workflow there are two Web severs, ra and optimus, in a Local Area Network. The CGI programs and associated HTML files will be archived (tarred) into two files, IZT_ra.tar, IZT_optimus.tar, depending on their ultimate destination. These tar files will be copied over and untarred into the specified cgi-bin directories on the respective Web servers. The destinations can be easily changed since WebWork supports location independence. This is achieved by treating the hostnames specified by the designer as logical names which are associated with the URLs for Web servers in a configuration file.

For any partitions of the application that need to be installed on a remote Web server, the process is less automated. This remote install process is initiated with the following command.

```
> make rinstall
```

The tar files will be created as before, but copying will be done manually with ftp. Once the tar files have been copied, the developer must remotely log into the servers and untar the files.

For more details on developing and installing workflow applications, please see the Workflow Developer's Guide [Pal97].

6 Conclusions and Future Work

The LSDIS lab at the University of Georgia is developing multiple implementations of the METEOR_2 model, OrbWork, NeoWork and WebWork. All will interoperate and work off of a common graphical designer. Each has its own particular strong points. This paper has focused on WebWork as an implementation of the METEOR_2 model which emphasizes ease of development of workflow applications, installation, and use. Since it uses Web technology, free compilers, and interfaces with common databases, most organizations need not purchase any base technology in order to run WebWork. Furthermore, it has the following desirable characteristics:

- WebWork supports the creation of flexible workflows that include both manual and automated tasks, transactional and non-transactional tasks as well as sophisticated, yet straightforward, ways of coordinating the overall execution of the tasks. These capabilities are prescribed in the METEOR$_2$ model.

- WebWork is designed for heterogeneous distributed environments spanning multiple organizations. Client access is universal since all that is required is a graphical Web browser. Even a low cost Network Computer could be used. On the server side where the CGI programs are run, WebWork supports the use of multiple Web servers. A variety of Web servers can be used and the underlying platforms supported are UNIX and Windows NT.

- WebWork is designed to interoperate with OrbWork, NeoWork and InfoSleuth [3] It can also communicate with a variety of DBMSs. In addition, it has facilities for wrapping legacy applications.

- WebWork provides low overhead, yet what we believe to be effective recovery mechanisms. We have classified ten different type of errors or failures that WebWork deals with.

- WebWork supports rapid workflow application development in which all workflow oriented code is automatically generated. A default screen is also generated. In addition, stubs for application tasks are generated so that a workflow application can be prototyped and tested with no coding required.

The philosophy behind the development of METEOR$_2$ implementations is that they should be built on top of powerful, yet commonly used infrastructures. The Web was chosen because of its ubiquitous, low cost and easy-to-use nature. CORBA was chosen because of its powerful distributed object management (e.g., it is able to support nested transactions in distributed and heterogeneous environments). One of the highlights of the METEOR$_2$ approach is that advanced transactional capabilities will be provided by the implementations. This requires at least the capabilities of a Transaction Processing Monitor (TPM) to be provided by the WfMS. Because of the complexity of this we are relying on the infrastructure to provide much of these capabilities (e.g., CORBA Object Transaction Services). The intent of WebWork is to provide most of the capabilities of its

[3] InfoSleuth is an agent-based information access facility developed by the Microelectronics and Computer Technology Corporation (MCC), one of our partners in the HIIT/HITECC project.

CORBA counterparts in a smaller and lower cost package. As such, it does not fully implement the $METEOR_2$ model. The following capabilities are missing.

- Transactional Subworkflows. Although individual tasks such as database updates may be transactional, a sequence or set of database updates on multiple databases, for instance, is not supported as transactional. This requires advanced transactional support (e.g., nested transactions, two-phase commit, sagas, etc.) from the WfMS.

- Advanced Recovery. Although WebWork deals with many types of errors and failures, it is not airtight. Certain, hopefully infrequent, failures may require complex manual interventions to restore the active workflows to a usable state.

- Complex Objects. OrbWork allows arbitrary CORBA data objects to be sent between tasks. The objects may have attributes, methods and relationships with other objects. WebWork does not support arbitrary data objects, but rather sends data tuples (analogous to the relational data model) between tasks. A data tuple consists a number of attribute values where each attribute values must come from an atomic (non-composite, and single-valued) domain. If, for instance, an image needs to be sent, its URL (a simple string) is included in the data tuple.

- Complex Data Dependencies. OrbWork allows multiple classes (object types) to be associated with a data dependency between tasks. At this time, WebWork restricts the association to a single class.

Although the WebWork implementation has a few limitations, we do not feel that they would prohibit successful deployment and use. This opinion is soon to be field tested, since WebWork 1.0 Alpha is being released to CHREF, our industrial partner. Next to be released will be OrbWork. This will allow us to compare the two approaches. Ultimately, we believe they will be complementary. Furthermore, as Web technology advances, it will begin to include some of the advanced capabilities of CORBA (e.g., reliable Web servers and atomic transactions). Later releases of WebWork will take advantage of these capabilities.

In the next release of WebWork, there are several new capabilities or enhancements that we wish to include. The list includes safe, yet easy to use security and authentication mechanisms, more advanced recovery mechanisms, a graphical monitoring tool, more sophisticated wrapping of legacy applications (possibly using Netscape's Constellation), use of reliable Web servers and atomic transactions, and finally, efficiency enhancements. Performance can be enhanced by using a cluster of Web servers with load balancing, better techniques for running tasks in parallel, newer http protocols, WebNFS, etc. Although these efficiency enhancements are of general interest to the Web community, we intend to focus on them from the perspective of workflow.

Acknowledgements

The METEOR team consists of Kemafor Anyanwu, Souvik Das, Prof. Krys Kochut (co-PI), Zhongwei Luo, Prof. John Miller (co-PI), Devanand Palaniswami, Kshitij Shah, Prof. Amit Sheth (PI), Harvinder Singh, Devashish Worah and Ke Zheng. Key past contributors include David Lin, Arun Murugan and Richard Wang.

This research was partially done under a cooperative agreement between the National Institute of Standards and Technology Advanced Technology Program (under the HIIT contract, number 70NANB5H1011) and the Healthcare Open System and Trials, Inc. consortium. See URL: *http://www.scra.org/hiit.html.* Additional partial support and donations are provided by Visigenic, Informix, Iona, Hewlett-Packard Labs, and Boeing.

References

[AAAM97] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionalities and Limitations of Current Workflow Management Systems. Technical report, IBM Almaden Research Center, 1997. To appear in IEEE Expert.

[ASSR93] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proc. of the 19th Intl. Conference on Very Large Data Bases*, pages 134–145, Dublin, Ireland, 1993.

[BSR96] A. Bonner, A. Shruf, and S. Rozen. LabFlow-1: A Database Benchmark for High Throughput Workflow Management. In *Proc. of the 5th. Intnl. Conference on Extending Database Technology*, pages 25–29, Avignon, France, March 1996.

[CGN97] T. Cai, P. Gloor, and S. Nog. DartFlow: A Workflow Management System on the Web using Transportable Agents. Technical report, Dartmouth College, 1997. URL: http://www.cs.dartmouth.edu/reports/Cai,Ting.html.

[Con97] Information Management Consultants. WebFlo - Delivery Imaging and Workflow over the Web. Technical report, Information Management Consultants, 1997. URL: http://www.imcinc.com/webflo.

[Cor97] WebFlow Corporation. Putting your team on the SamePage. Technical report, WebFlow Corporation, 1997. URL: http://www.webflow.com/product.html.

[CS897] CS835. Enterprise Integration Course Technical Reports. Technical report, University of Georgia, Department of Computer Science, 1997. URL: http://lsdis.cs.uga.edu/ amit/EI.html.

[DKM+97] S. Das, K. Kochut, J. Miller, A. Sheth, and D. Worah. ORBWork: A Reliable Distributed CORBA-based Workflow Enactment Systems for METEOR_2. In *Proc. of the 23nd. Intnl. Conference on Very Large Data Bases*, Athens, Greece, August 1997. (under review).

[Dog96] A. Dogac. Special-Theme Issue: Multidatabases. In *Journal of Database Management*. Idea Group Publishing, Harrisburg, PA, Winter 1996.

[Fis95] L. Fischer. *The Workflow Paradigm - The Impact of Information Technology on Business Process Reengineering*. Future Strategies, Inc., Alameda, CA, 2nd. edition, 1995.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–154, April 1995.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Hol94] D. Hollingsworth. The Workflow Reference Model. Technical Report TC00-1003, Issue 1.1, The Workflow Management Coalition, Brussels, Belgium, November 1994.

[JAD+94] S. Joosten, G. Aussems, M. Duitshof, R. Huffmeijer, and E. Mulder. *WA-12: An Empirical Study about the Practice of Workflow Management*. University of Twente, Enschede, The Netherlands, July 1994. Research Monograph.

[JK97] S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997. To be published.

[KL96] G. Kaiser and Wenke Lee. Pay No Attention to Man Behind the Curtain. Technical report, Columbia University, 1996. URL: http://www.psl.cs.columbia.edu/papers/nsf96.html.

[KS95] N. Krishnakumar and A. Sheth. Managing Heterogeneous Multi-system Tasks to Support Enterprise-wide Operations. *Distributed and Parallel Databases*, 3(2):155–186, April 1995.

[MSKW96] J. A. Miller, A. P. Sheth, K. J. Kochut, and X. Wang. CORBA-based Run-Time Architectures for Workflow Management Systems. *[Dog96]*, 7(1):16–27, Winter 1996.

[NAS96] NASA. Program Information Systems Mission Services (PrISMS). Technical report, NASA, 1996. URL: http://ec.msfc.nasa.gov/prisms.

[Pal97] D. Palaniswami. Development of WebWork: METEOR$_2$'s Web-Based Workflow Management System. Master's thesis, University of Georgia, Athens, GA, June 1997. In preparation.

[PLS+96] D. Palaniswami, J. Lynch, I. Shevchenko, A. Mattie, and L. Reed-Fourquet. Web-based Multi-Paradigm Workflow Automation for Efficient Healthcare Delivery. In *[She96]*, Athens, GA, May 1996.

[RS95] M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*. ACM Press, New York, NY, 1995.

[SGJ+96] A. Sheth, D. Georgakopoulos, S. Joosten, M. Rusinkiewicz, W. Scacchi, J. Wileden, and A. Wolf. Report from the NSF Workshop on Workflow and Process Automation in Information Systems. Technical report, University of Georgia, UGA-CS-TR-96-003, July 1996. URL: http://LSDIS.cs.uga.edu/activities/NSF-workflow.

[She96] A. Sheth. Proc. of the NSF workshop on workflow and process automation in information systems. University of Georgia, May 1996. URL: http://LSDIS.cs.uga.edu/activities/NSF-workflow.

[SJ96] A. Sheth and S. Joosten. Workshop on Workflow Management: Research, Technology, Products, Applications and Experiences, August 1996.

[SKM+96] A. Sheth, K. J. Kochut, J. Miller, D. Worah, S. Das, C. Lin, D. Palaniswami, J. Lynch, and I. Shevchenko. Supporting State-Wide Immunization Tracking using Multi-Paradigm Workflow Technology. In *Proc. of the 22nd. Intnl. Conference on Very Large Data Bases*, Bombay, India, September 1996.

[Tec97] Action Technologies. Metro 3.0 Coordinates Work across the Web. Technical report, Action Technologies, 1997. URL: http://www.actiontech.com/Metro.

[Wan95] X. Wang. Implementation and Performance Evaluation of CORBA-Based Centralized Workflow Schedulers. Master's thesis, University of Georgia, Athens, GA, August 1995.

[WS97] D. Worah and A. Sheth. Transactions in Transactional Workflows. In *[JK97]*, chapter 1. Kluwer Academic Publishers, 1997.

[Zhe97] K. Zheng. Development of the $METEOR_2$ Designer. Master's thesis, University of Georgia, Athens, GA, June 1997. In preparation.

Appendix

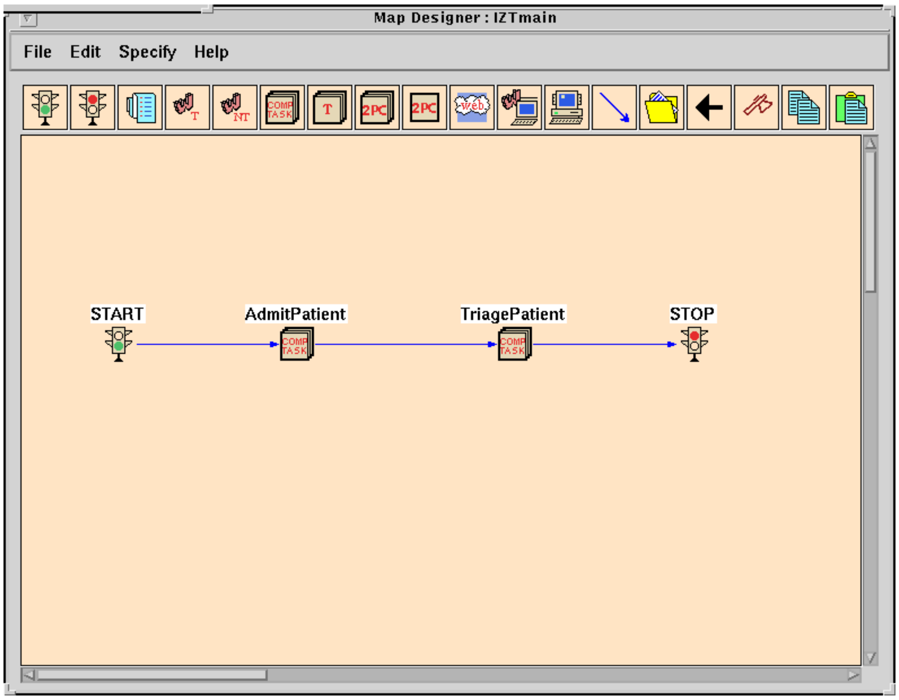

Figure 8: Immunization Tracking Workflow Design

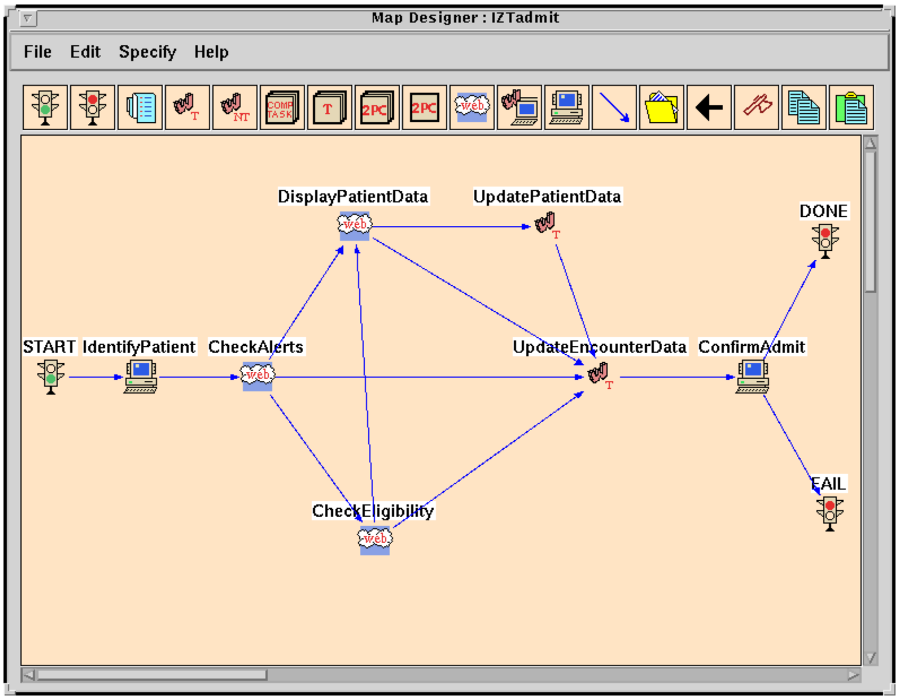

Figure 9: Admit Clerk Sub-Workflow Design

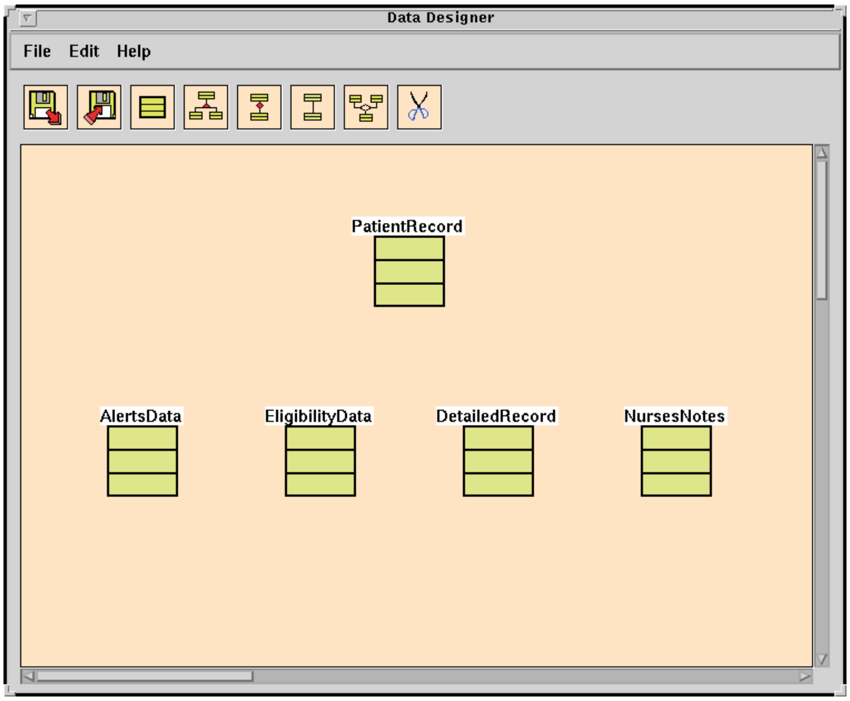

Figure 10: Immunization Tracking Data Design

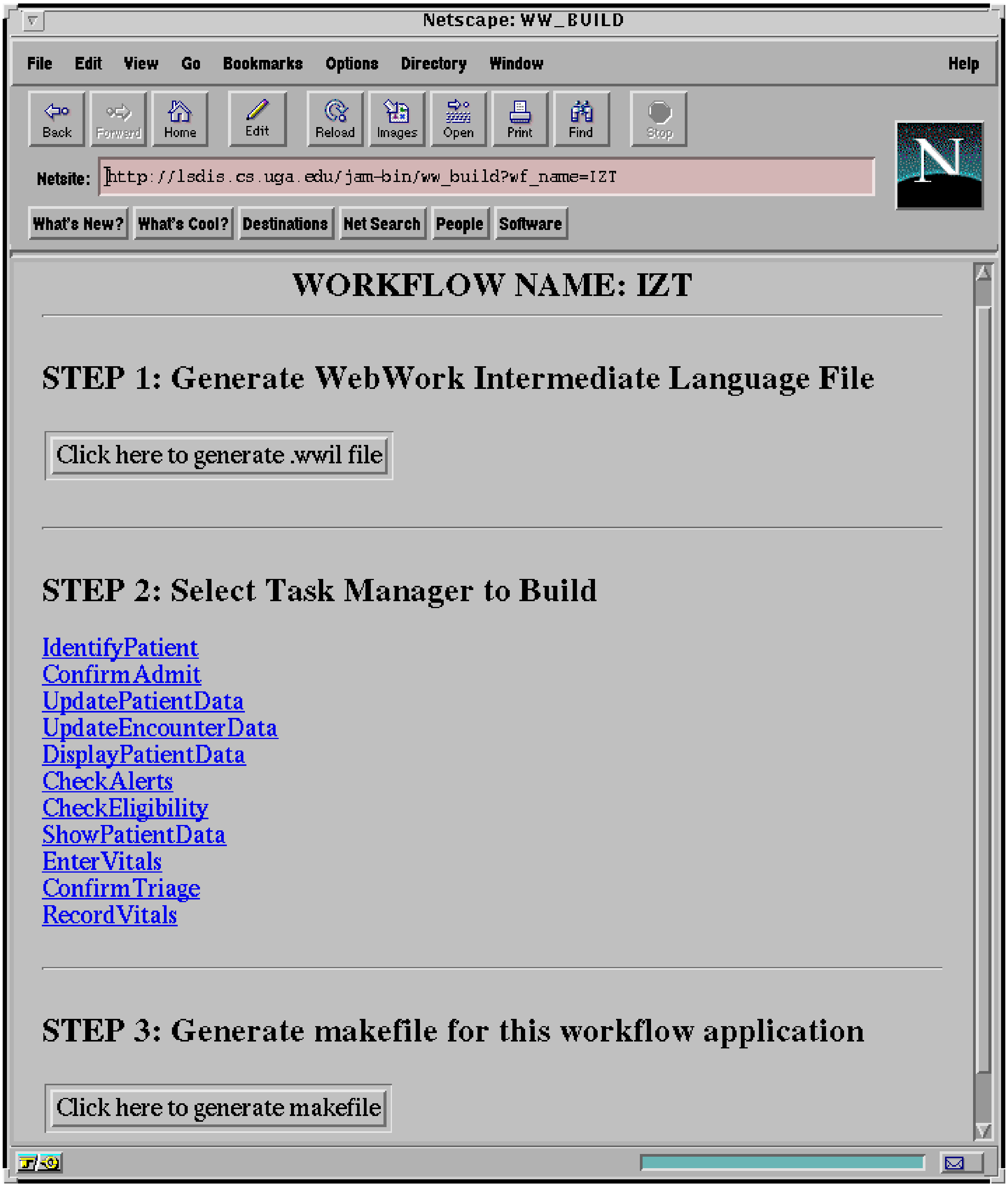

Figure 11: WebWork Builder

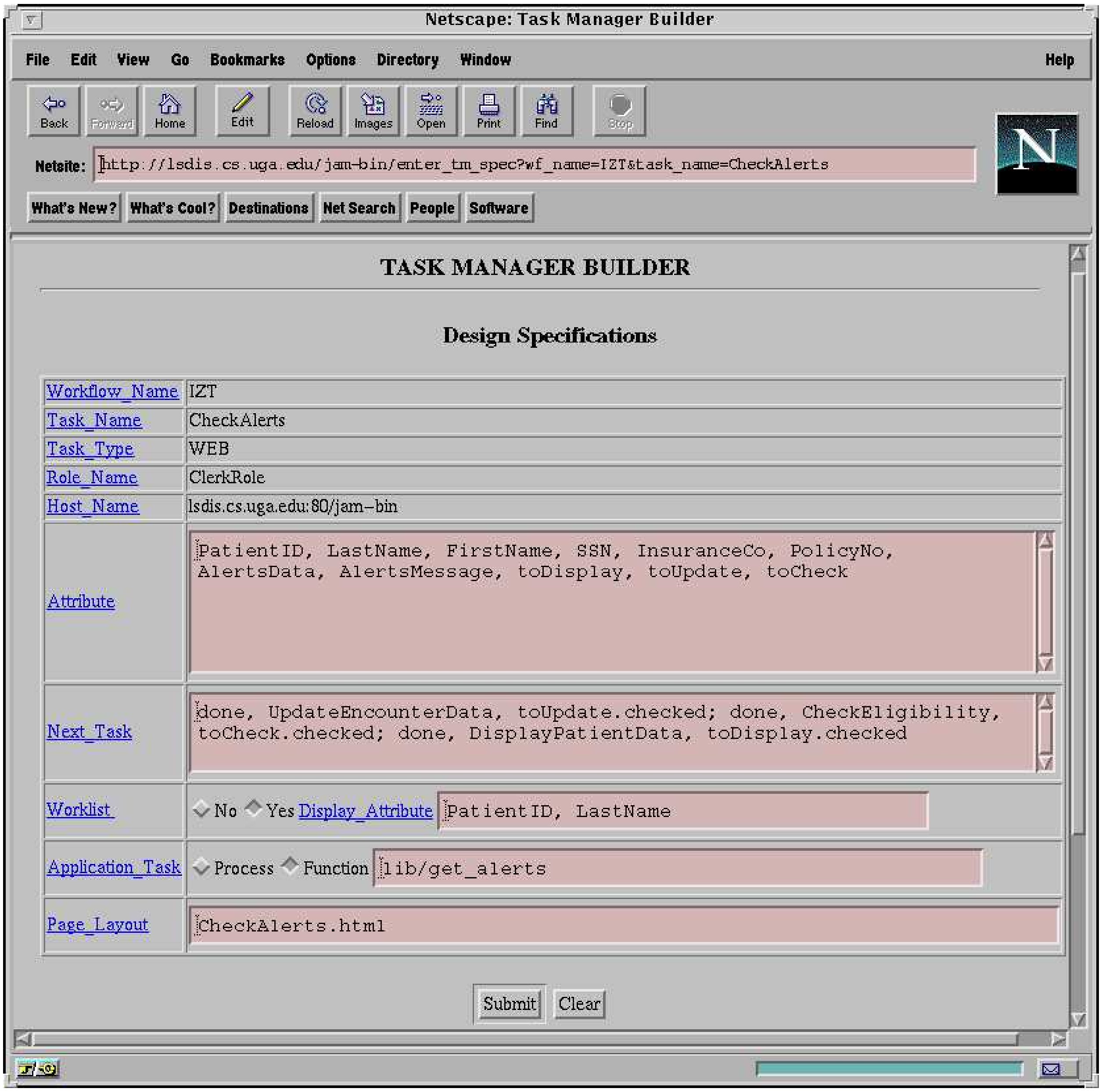

Figure 12: Check/Customize Task Specification

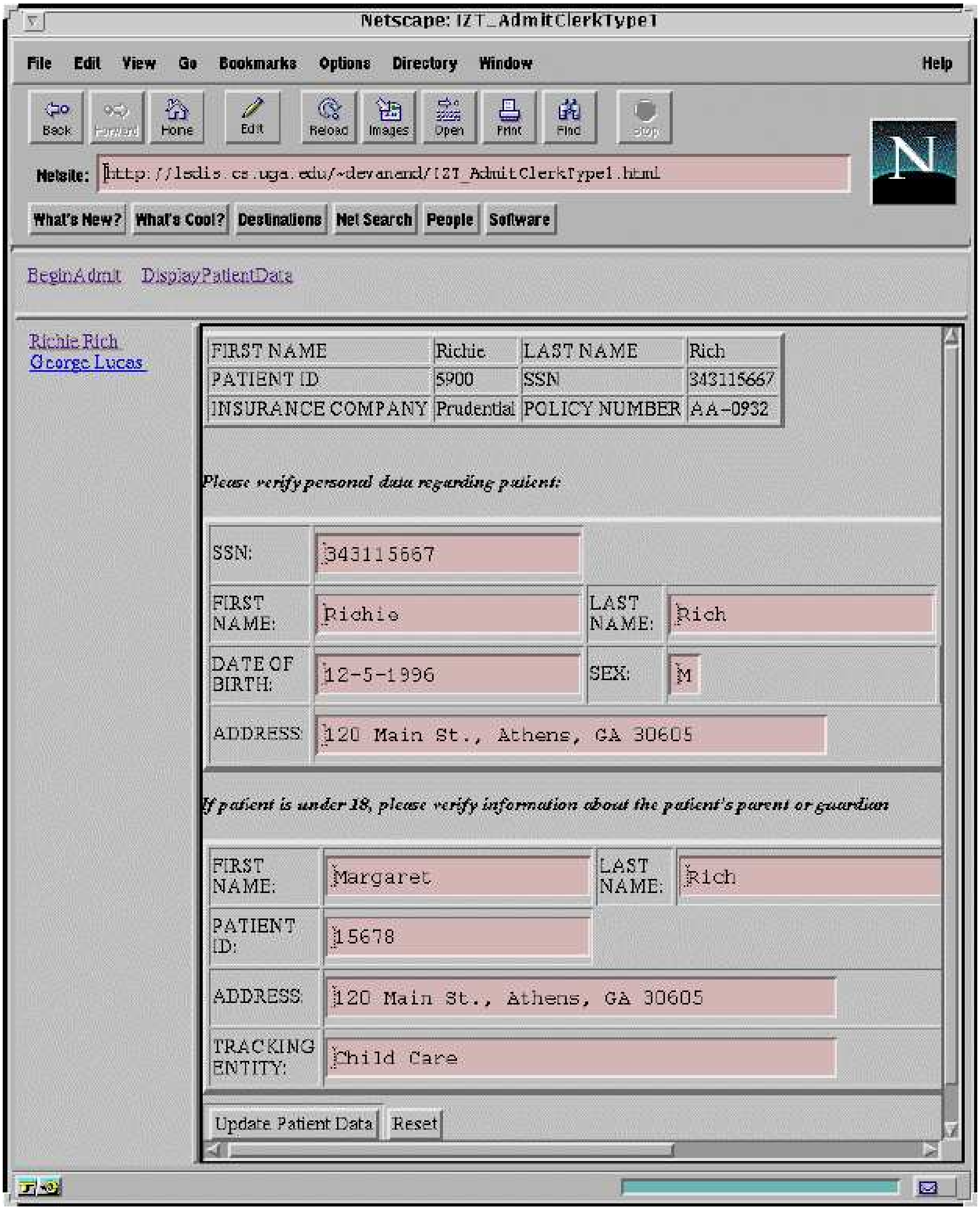

Figure 13: Screenshot of DisplayPatientData Task