An Architecture for WWW-based Hypercode Environments

Gail E. Kaiser **Stephen E. Dossick**
Wenyu Jiang **Jack Jingshuang Yang**
Columbia University
Department of Computer Science, MC 0401
New York, NY 10027, UNITED STATES
212-939-7081, kaiser@cs.columbia.edu

CUCS-037-96, 8 August 1996

ABSTRACT
A *hypercode* software engineering environment represents all plausible multimedia artifacts concerned with software development and evolution that can be placed or generated on-line, from source code to formal documentation to digital library resources to informal email and chat transcripts. A hypercode environment supports both internal (hypertext) and external (link server) links among these artifacts, which can be added incrementally as useful connections are discovered; project-specific hypermedia search and browsing; automated construction of artifacts and hyperlinks according the software process; application of tools to the artifacts according to the process workflow; and collaborative work for geographically dispersed teams. We present a general architecture for what we call hypermedia *subwebs*, and *groupspace* services operating on shared subwebs, based on World Wide Web technology — which could be applied over the Internet or within an intranet. We describe our realization in **OzWeb**.

Keywords
Hypermedia, Environments, Software process, Workflow, CSCW, Software documentation

INTRODUCTION
Software engineering environments seek to improve quality and productivity, a major concern for long-lived software systems and families of systems. *Hypercode* environments intertwine design, implementation and evolution, cross-referencing and assisting users in generating, retrieving, updating and exploiting all on-line materials that may plausibly be relevant to a project. Without hypercode, available documentation may not include all the nitty details that came up during meetings, email, discussion groups, etc. The apparent minutia, some of which may later be quite valuable for desiderata and rationale, has likely been lost.

Programming language code and formal prose documents, flow diagrams, etc. may make up only a small proportion of hypercode. There could also be arbitrary informal documents such as email and newsgroup archives, meeting reports perhaps including video and audio components, even scanned-in diagrams sketched on napkins, as well as traces of the process that has been followed to date during development and evolution, with various analyses and metrics regarding code, documents, and process. See [7] for further rationale.

The hyperlinked nature permits references to materials generated and managed *independently* from the project, such as digital library publications, technical reports from unrelated institutions that provided algorithms, architectures or insights, videos or transcripts of lectures with some bearing on the system, etc. Thus we argue that to make hypercode environments as useful as possible, the materials should not reside in a single repository internal to the project or the organization, but must potentially be able to contain links out to external sources and, when desired, permit incoming links from other organizations. Users must be able to impose links on top of and orthogonal to any hyperlinks embedded in documents by the authors, to add references to materials the authors never thought of or that did not exist when the document was written.

This is very nearly what the World Wide Web (WWW) provides. Thus we choose Web technology, particularly Hypertext Transfer Protocol (HTTP) and Hypertext Markup Language (HTML), as the infrastructure on which to construct hypercode environments. Web technology can be applied within an intranet, with outgoing links tunneled through corporate firewalls; it is **not** necessary to make proprietary software engineering materials publicly available on the Internet.

However, the Web alone, or in tandem with other distributed computing infrastructures (e.g., DCE, OLE, CORBA), does not provide the *services* necessary for software engineering environments:

There is little *organization* of Web documents, a great advantage for generic use, but consequently project-specific navigation and search are challenging. There is no standard means to *use* information once found. To treat the Web as a repository for software engineering environments, it is necessary to apply both conventional tools (analyzers, compilers, debuggers, etc.) and emerging Web-aware tools, and place their results back onto the Web. Common Gateway Interface (CGI) and Java can be used for tool scripts, but some general mechanism is needed for *managing* tool execution.

There is no support for software development *process* (or workflow) modeling and enactment regarding access to Web pages, to present users with, guide and assist them in following a process designed to support development and evolution of high quality software products at low cost. Software process has been a central concern within the community for more than a decade. Although the Web may soon support versioning and configuration management following the "checkout model", there is little concept of *transactions*, with either the classical atomicity, consistency, isolation and durability properties, or permitting relaxations proposed for long-duration, interactive, and/or cooperative work [10, 6]. The first author argues in [11] why the checkout model is less attractive than "cooperative transactions".

We propose an innovative architecture that makes it relatively easy to add organization, process, transactions and related services to the World Wide Web infrastructure: Our approach does **not** require a special browser, *any* browser that supports the HTTP 1.0 standard is sufficient. Our approach does **not** require use of a special server, *any* HTTP 1.0-compliant web server will do. The "trick" is to use HTTP proxy servers, which can mediate all traffic from/to any browser with respect to any servers. Our sample graphical user interface (GUI) assumes the HTML 2.0 standard (e.g., frames), but other GUIs could be designed for the same architecture.

We have implemented a prototype hypercode environment framework called **OzWeb**, and a working environment instance. This research is concerned with how to apply project-specific organization and search, process, and cooperative transactions to Web materials, **not** which specific data definition notation, process modeling language and enactment model, concurrency control mechanism, etc. might be best for software engineering environments. Thus we adapted these facilities from the Oz process-centered environment [3] to our hypercode architecture, rather than attempting to invent new ones or arguing in favor of our old ones.

A MOTIVATING SCENARIO

The Spring 1996 Introduction to Software Engineering course at Columbia involved a team project where groups were designed, coded and tested a query processor for a given object-oriented database system (the Darkover OODB prototype developed in our lab). The objectbase implementation was provided as C header files and object code libraries, and a functional specification for the ideal query processor was given. Each group developed a structured design for the query processor. The design documents were swapped with other groups, who performed design review and then coded this design. The groups inspected and revised their own code, and swapped with another group for testing. Since only two groups had complete implementations, the "lobster" group tested the code from "whitespace" and all other groups tested the "lobster" code. Some groups submitted their homework in HTML or postscript format from their Web pages.

The lobster students were invited to take the Undergraduate Projects in Computer Science course in Summer 1996, to extend their query processor. Several graduate students and a staff member were concurrently working on the same OODB, and on the Oz and **OzWeb** systems constructed on top; everyone was working against tight deadlines. This was **not** "homework", they were developing a significant piece of our system — but will soon graduate and disappear, so their code will be maintained by other people.

We have numerous potentially *useful* artifacts on-line: The OODB manual, source code, interface and library; class lecture notes, assignments, readings, etc.; the functional specification for the assigned query processor and the student proposal for its extensions; design documents from several groups; baseline code (at semester end) and new code (in progress) from the lobster group; lobster's code inspection reports; the sample data model and objectbase supplied for testing; testing reports and journals from other groups on lobster's code; email between the students and instructor; and the class newsgroup archive. One could also imagine talk/chat transcripts, and video and/or audio meeting records.

Simply placing these materials on-line is inadequate. Manually creating and maintaining cross-references as the work continues and documents change would be an enormous undertaking. Instead, *useful* hyperlinks should be automatically generated when possible, and be automatically added as the materials undergo further evolution. Environment users should be able to introduce hyperlinks incrementally as they discover *useful* relationships. We emphasize *useful*, because simply indexing or cross-referencing every appearance of a given word or phrase would result in a densely linked mass of little value. Instead, we advocate a "loose" schema categorizing and organizing entities of interest, process modeling to indicate semantic dependencies, and workflow automation to automatically add/remove hyperlinks as tasks are completed, as key value-adding services. Concurrency control and recovery are also important.

APPROACH

Subwebs

A *subweb* organizes on-line materials of plausible interest to a project, such as our motivating scenario, over its life-time. A subweb supports structured associative and navigational queries, and unstructured information retrieval and hyperlink following. Some materials may be updated for unrelated purposes, so it is not always appropriate to copy everything to a project-specific repository: some documents should continue to reside at their original homes but be treated as part of the project's information base, perhaps without the author's knowledge (for publicly available materials). It should also be possible to retain private copies of selected documents.

Subweb organization follows what we call a "loose" schema because it specifies the existence, names and types of possible composition and referential links imposed on top of the materials, as well as other attributes, but does **not** define or affect any hyperlinks that may be embedded in their content. In particular, the schema may introduce *external* links among documents, images, etc. that are unknown (and perhaps irrelevant) to the authors of those materials. Thus subweb is a form of *link server* with an object-oriented database veneer, in which those objects that are instances of, say, the `WebObj` class correspond roughly to link anchors such as WWW Universal Resource Locators (URLs). Anchors need not refer to full pages or images, but may correspond to a point within a document, e.g., using the fragment name feature of HTML (`<A HREF="...#<name>"` and `<A NAME="...">`) or to a particular state of a tool applied to the anchor as in Chimera [1].

Figure 1 shows a sample `WebObj` class and two subclasses. `WebObj` represents the anchor (i.e., a fully-qualified canonical `URL`) as an ASCII string; defaults to `Reference` as the access `type`; treats the hypermedia entity `content` as opaque (binary); and stores related information like timestamp and location in a text file. The `my_links` and `my_children` attributes allow arbitrary references and composition of objects.

The *Reference* access type refers in virtual form to whatever entity is found by accessing the anchor. The entity might be changed "out from under" the subweb, or become inaccessible at any time (although the `content` field is used as a cache). Repeated accesses (e.g., HTTP conditional GET) might not retrieve the identical document. All accesses via the subweb are read-only. *Updatable* is the same as reference, except that subweb users can modify the entity (e.g., using HTTP PUT). The subweb may "own" the website, for storing project-specific materials, or an agreement might be made between website and subweb administrators. Process constraints are employed, so not necessary *all* users of a given subweb can make updates, perhaps only specially

```
WebObj :: superclass ENTITY;
    URL : string;
    type : (Reference, Copy, Updatable)
        = Reference;
    content : binary = ".html";
    updateinfo : text = ".updinfo";
    my_links : set_of link ENTITY;
    my_children : set_of ENTITY
end

TEST_REPORT :: superclass WebObj;
    tester : user;
    bug_src : set_of link CFILE;
end

CFILE :: superclass FILE, WebObj;
    source : text = ".c";
    object : binary = ".o";
    lock_status : (Available, CheckedOut)
        = Available;
    compile_status : (NotCompiled,
        ErrorCompiled, Compiled) = NotCompiled;
    test_reports : set_of link TEST_REPORT;
end
```

Figure 1: WebObj Superclass and Sample Subclasses

privileged users, or user roles, and then only when particular prerequisites are satisfied. *Copy* refers to a subweb-resident copy, made when the web object is first instantiated, or re-instantiated on demand later on (e.g., a Web crawler might periodically check for modifications and notify users). The copy can be modified by subweb users (although a particular hypercode environment might add a "read-only" attribute enforced by process constraints), but is **not** intended as a write-through cache: changes to and new versions of the copy do not in any way impact the original.

In addition to schema-based queries, subwebs provide text-matching search analogous to yahoo, lycos, altavista, etc. But search is *restricted* to only those hypermedia documents represented by a selected set of web objects (perhaps the entire subweb), or a subset of the documents reachable via embedded links from the web objects (e.g., on the same website). Current Web search engines usually index everything they can find, or restrict to a particular website, and are often present many screenfuls of seemingly random junk.

When a user reads a web object from a subweb, the entity's content is augmented to display: an icon anchor whose selection brings up the subweb GUI on the user's screen in another browser window; the values of primitive attributes defined by the web object's class and its superclasses; anchors corresponding to each file, composite and reference attribute (multiple anchors for set values); and additional subweb-specific materials, e.g., there might be a "message of the day" to all subweb users plus user-specific notices.

The user may write the entity, e.g., using an HTML editor or tool output, only if the target web object is `Copy` or `Updatable`. For `Copy`, the local copy in the web object `content` attribute is replaced (or a new version created); for `Updatable`, both the `content` cache and the original are modified (e.g., via PUT). If the URL points to a directory or otherwise results in generation of an HTML page, that page is treated as the `content`, but writing may not be meaningful.

WWW supports *forms* whereby the user enters input to arbitrary backend programs, to be transmitted and processed via the CGI protocol (HTTP POST). There are several user interaction models: The user might intend the web object to represent the blank form (to submit arbitrary queries later), or the filled-in form (effectively a particular query that might be resubmitted later with different results), or the output of submitting the form with particular inputs (the result of a previous query), with different read/write implications.

Groupspaces

To construct fully functional software engineering environments on top of hypercode subwebs, we add what we call *groupspace* services consisting of multi-participant software process workflow automation, concurrency control and recovery for collaborative work, and tool management technologies. We allow for addition of other services, such as content-based security mechanisms and general access control, not explored here. Some or all groupspace services could be omitted in a given implementation, and subwebs would still be useful for organizing materials. We choose the term "groupspace" because we think of each affected Web browser as the user's personal workspace, and collaboration support through these services turns the relevant portion of the Web into a cooperative forum.

Any process paradigm (Petri nets, rules, task graphs, etc.) can be adapted to hypercode. We assume here that a process model includes partially ordered tasks, allowing for alternatives and iteration. Tasks usually have parameters involving process state and/or product artifacts, may involve invocation of external tools, may form a hierarchy of subtasks, may have prerequisites and consequences, and may specify resource needs. Parallel work by human participants may be synchronized.

Once a subweb mechanism is in place, process modeling and analysis needs relatively little extension to work with hypercode. We identify two main concerns: treating hypermedia accesses as tasks on which process constraints may be applied, and dependencies among data items that should be reflected as hyperlinks.

The process notation should allow *read* and *write* of a hypermedia entity, *follow* of an embedded or external hyperlink (*follow* might not be distinguished from *read* where the user selects the desired entity explicitly), and *link/unlink* of an external hyperlink, to be treated as primitive tasks. Detecting *link/unlink* for embedded hypermedia content may also be desired, but is likely to be more costly in performance. There may be additional primitive tasks such as *add*, *delete*, *copy*, *move*, *rename*, etc. Composite hypermedia operations, such as reading or writing a compound document consisting of multiple entities, might also be recognized as process tasks.

Then partial ordering, synchronization, prerequisites and consequences, etc. should be applied to these operations in the same manner as for the native tasks supported by the process system. For example, prerequisites must be satisfied before a browser may successfully retrieve (GET) or update (PUT) the WWW entity associated with a web object. No prerequisites are enforced when accessing URLs (e.g., a friend's home page or www.olympic.att.com) outside the subweb. This extension to process concepts may not involve significant changes, if the original language already includes *read* of a project datum, etc. In a hypercode environment, these operations apply to web objects as objects, and to their underlying hypermedia entities, as well as to other data. An ideal subweb implementation would make this distinction completely transparent to the process engine.

The process notation should also include facilities for specifying those dependencies among entities where composite and/or referential links are required to be explicitly placed under human supervision vs. those links that could/should be inferred and implicitly placed via process automation. For instance, a task prerequisite may require that a particular link exist between two of its parameters, or an implicit parameter might be derived by following a specified link (whose absence might result in failure to satisfy the prerequisite, depending on the logic written by the process designer). Or a task consequence may introduce or remove a link between its inputs (parameters and derived parameters) and outputs, including parameters derived via associative queries or information retrieval.

Such prerequisites and consequences could be defined manually by the process designer, to carefully determine the cases where hyperlinks should be placed by a cognizant user vs. automatically inserted. Or the process model might be mechanically transformed according to template(s) constructed by the process designer or provided by the hypercode environment framework, although freewheeling automation may result in a dense and useless mass of cross-links. This extension to process concepts may not involve any syntactic or semantic changes, if the original language already includes such support (operating on its native data repository). Again, the subweb should hide the distinction between web objects and native data from the process engine.

Enactment of the groupspace-specific process, and visualization of the process while it is in progress, requires that all Web browser requests be *intercepted* so that prerequisites, consequences, synchronization, etc. are enforced and automated, tasks and task segments operating on WWW entities are automatically initiated and controlled, users are notified when tasks they are supposed to do become enabled or tasks that affect them are completed, and visualization display(s) are updated as the process unfolds. Different process systems may not provide all these capabilities, e.g., some process systems attempt to automate satisfaction of prerequisites and assertion of consequences and thus might introduce appropriate hyperlinks without human intervention, whereas others support more limited workflow that only prompts the human to do the work. For all these functions, interception of all HTTP traffic emanating from a participating Web browser is mandatory.

If the process formalism is adapted to cover changes to hypermedia *content*, particularly `Reference` and `Updatable` web objects that change independently of the subweb, then a polling scheme is needed, e.g., a periodic Web crawler that notifies affected subwebs (we cannot assume a notification capability by the website itself, since subwebs and groupspaces must work with *all* HTTP servers). Such notification must trigger proper process enactment, perhaps without an attending human user. Some process systems may already handle external changes (outside their control) to their data repositories or other “off-line” invocation.

Process measurement and evolution do not seem particularly complicated by hypercode — except that the underlying entities may be manipulated outside the subweb mechanism. Thus it is virtually impossible for groupspaces to maintain fully accurate statistics or guarantee that any process state embedded in WWW components change only through controlled evolution. This is more insidious that the inherent problem where a user becomes “root” and arbitrarily manipulate a project repository through operating system facilities, because WWW hypercode has no central authority.

Groupspaces do not require any particular transaction model for collaborative work, but there must be some model — preferably one that takes process semantics into account to permit appropriate serializability conflicts (e.g., when using groupware tools) and avoid in-

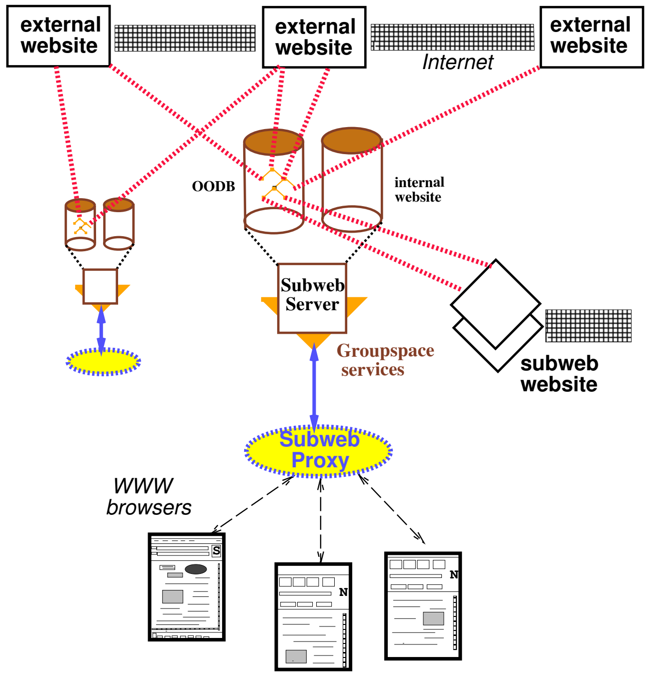

Figure 2: Subweb Architecture

appropriate rollback of partial changes interrupted by failures (e.g., when human labor vs. regeneratable tool output is at stake). To enforce that two groupspace users are not updating the same WWW entity or related entities at the same time, or over writing each others' changes when concurrency control policies disallow it, the system must again *intercept* all Web accesses.

Finally, any software engineering environment must support tools, in the hypercode case both commercial off-the-shelf (COTS) tools with no notion of hypermedia and emerging Web-aware tools (e.g., HTML editors and mailers). If Web-aware tools can be invoked directly from the operating system, the subweb implementation should still guarantee that the tool will receive the appropriate view, e.g., the subweb's local copy if the web object is of type `Copy`, as opposed to the original entity from its home website. Otherwise, tools can easily (and unintentionally) subvert groupspace services. If a non-Web tool is invoked outside the groupspace interface, say directly on the underlying file system, it is generally impossible to impose groupspace services or subweb discipline except through access control.

ARCHITECTURE

Figure 2 illustrates our subweb architecture. The key component is the *subweb proxy*, through which all Web traffic is funneled. Two distinct subwebs are shown, call them Little (the one on the left) and Big (the one on the right). The Big subweb contains six web objects whose URL attributes point to pages located one at website A, two at website B, one at website C, and two at the subweb's own websites. Subweb users are depicted as WWW browsers. The browsers are configured to transmit all their traffic through the subweb's proxy server. These may be daisy-chained with other proxy servers before and after the subweb proxy, performing other functions, as in OreO's stream transducers [4]. The same subweb could be accessed through multiple subweb proxies. The browsers, proxy server(s), subweb server, subweb website(s) and any other relevant websites may reside on different machines dispersed across the Internet or an intranet.

When one of these browsers sends a GET request for a URL, the subweb proxy asks its subweb server if that URL corresponds to any web object. If not, the subweb proxy behaves like a conventional HTTP proxy and contacts the website server indicated in the URL to retrieve the entity, which is sent to the browser in the normal fashion. The only change is that the proxy server also sends along a second HTML frame with its the groupspace service logo and some menu items that may be relevant to non-subweb documents, e.g., a command to add the current document to the subweb (and determine its canonical URL, since it might have been referenced relative to some other document). The proxy server and subweb lookup add a small overhead to each access, negligible compared to network delay, on the same order of 20 milliseconds as for a caching proxy.

If the requested URL matches a web object, the subweb server sends it to the proxy. The mechanics of subweb server retrieval depends on the access type, i.e., it contacts the originating website for a `Reference` or `Updatable` web object but supplies its private copy for `Copy`. The proxy server should tack on additional materials (groupspace icon, etc.) only for those browsers listed in the proxy server's configuration file, to be matched against the HTTP `User-Agent` field; the point is to support arbitrary Web-aware tools, which would not be listed, to operate within the subweb discipline but receive only expected entity. Ignoring GUI use of frames, which adds an HTTP Redirect exchange, the basic protocol is shown in Figure 3.

At the subweb server, the retrieval is sandwiched between groupspace services: process and concurrency control constraints are applied first to determine whether or not the `read` (or `write`) operation can succeed at this time and by this user in this context. The operation might be rejected if the user does not have proper authorization with respect to the given web object, if the process prerequisites have not been satisfied (and cannot be automatically satisfied through process automation), or if there is a concurrency control conflict with another user who is concurrently accessing that object as part of a transaction. After sending the document to the subweb proxy (and hence to the user),

```
Browser sends "GET <URL>" to proxy server.

Proxy server sends "GET /QueryAttr/<URL>"
to subweb server.

If the subweb contains the <URL>
    Then the subweb server sends "HTTP/1.0 200
    OK\nContent-type: text/html\n\n
    <content>" to proxy server.

    Proxy server sends the same thing on
    to browser.
Else if not
    Then the subweb server sends "HTTP/1.0 404
    Not Found" to proxy server.

    Proxy server performs standard "GET <URL>".
End if
```

Figure 3: Subweb Proxy Server Protocol

the subweb server performs process-specific operations indicating the consequences of the `read`, which triggers process automation to fulfill the implications.

Note this is different from the usual process enactment model, where the effects are asserted only *after* the task has been completed: It is impossible to determine in the Web browser case whether or not the user is "done" with his/her reading. Although another icon could be added to the document display, which the user would click to indicate completion, this cannot be taken as proof that the user will not return to reading the document later on without notifying the subweb server since the document could be stored in the browser's own cache. There is a similar issue regarding concurrency control, since arbitrary browsers do not signal when reading is "done".

OZWEB REALIZATION

Oz Background

Oz [3] provides a rule-based process modeling language in which a rule generally corresponds to a workflow step. A rule specifies the step's name as it would appear in a user menu or agenda; typed parameters and bindings of local variables from the project objectbase; a condition that must be satisfied before initiating the activity to be performed during the step; the tool script and arguments for the activity; and a set of effects, one of which asserts the actual results of completing the activity. Built-in operations like `add`, `delete`, etc. are modeled as rules, and can be overloaded, e.g., to introduce type-specific conditions and effects on those operations; built-in operations can also be used in the effects of other rules.

Oz enforces that rule conditions are satisfied, and automates the process via forward and backward chaining. When a user requests to perform a step whose condition is not currently satisfied, the system backward chains to execute other rules whose effect may satisfy the condition; if all possibilities are exhausted, the user is informed that the chosen step cannot be enacted at this time. When a rule completes, its asserted effect may trigger automatic enactment of other rules whose conditions have become satisfied. Users usually control the process by selecting rules representing entry points into composite tasks consisting of one main rule and a small number of auxiliary rules (reached via chaining), but it is possible to define complete workflows as a single goal-driven or event-driven chain.

Oz employs a client/server architecture. Clients provide the user interface and invoke external tools. Servers context-switch among multiple clients, and include the process engine, object management, and transaction management. An Oz environment usually consists of several servers, each with its own process model, data schema, objectbase and tools. Clients are always connected to one "local" server, and may also open and close connections to "remote" servers. Servers communicate among themselves to establish and operate *alliances* supporting process interoperability.

OzWeb Implementation

OzWeb extends the Oz server to operate as a subweb server — and also as a general-purpose HTTP server (ftp, gopher, etc. protocols are not yet supported). The subweb proxy is a client of the **OzWeb** server, communicating with it via HTTP. Both HTTP components were implemented using our ASHeS toolkit (Application Specific Http Service), which can be used to construct arbitrary HTTP servers and proxy servers and turn existing systems into HTTP-compliant clients and servers. Native Oz clients continue to work with the **OzWeb** server, and are useful for debugging. The user browsers, proxy server(s), subweb server, subweb website(s) and any other websites represented in a subweb may all reside on different machines dispersed across the Internet, but could also reside on the same host. **OzWeb** subweb and proxy servers run on Solaris 2.5, with browsers and websites on any mix of platforms.

The subweb server is implemented using the same objectbase from the Motivating Scenario. When the proxy asks the subweb server whether or not a given URL is represented in the subweb, an objectbase query is formulated to search for a web object with that URL. The objectbase also serves as a website directory tree. The subweb server implementation is a separate layer, so that another OODB could potentially be substituted.

Oz's process engine was extended with *read, write* and *search* operations that can be overloaded by process-specific rules. The subweb mechanism transparently supplies the `content` attribute for web objects, or another file attribute (or all of them, in the case of *search*) for other objects. These could alternatively have been implemented purely as process-specific rules, rather than built-in operations, but this approach permits us to use them in the effects of arbitrary rules and is more efficient. Oz already provided *link*, *unlink*, *add*, *delete*, etc. operations that can be overloaded by process tasks. We decided that the process designer must overload the *link* built-in with a rule to generate the inverse link (in the objectbase), if desired, rather than hardwiring this behavior into **OzWeb**, so that only relationships deemed *useful* are explicit and thus displayed (as HTML links) during browsing. We have not yet implemented a polling mechanism for detecting externally-generated changes inside web object content. Oz's transaction manager was not changed.

Any HTTP-compliant browser (or other tool) works as described in the general approach above. When a registered browser receives any WWW entity (whether in the subweb or not), another frame is sent to display an **OzWeb** panel with an icon to select the **OzWeb** control screen, shown in another browser window. This gives the full Action Menu, and a two-column table with a list of the names of objects at the "current" level on the left hand side and the names of the children of the currently selected object on the right hand side, all represented as HTML links. Selecting a child moves it to the left and shows its children on the right; there is also a link for moving up in the objectbase composition hierarchy. The current position in the objectbase is encoded in the URL transmitted to the browser (e.g., `"http://<website>/ObjectView/OID"`) so that the subweb proxy server need not retain any memory of one browser client vs. another. Applying the *read* operation to any of these objects shows its `content` (if a web object, otherwise some file attribute must be selected) and other attributes in a browser window; the implicit *print* operation always displays the set of attributes (and primitive values) of the currently selected object. Selecting any link in the Action Menu brings up browser windows for entering arguments. When an argument may be an object, a browsing table is shown; textual entry (e.g., for *search* patterns) is also supported.

The GUI objectbase browsing tables contain URLs generated by the **OzWeb** subweb server rather than some conventional website, but the content of those objects need not be local. So the URL that the browser sees may not be the same as the entity's home URL; this is a subtle issue (and introduces the possibility of infinite loops and/or accidently overwriting if not handled carefully). Selecting an HTML link from the GUI thus

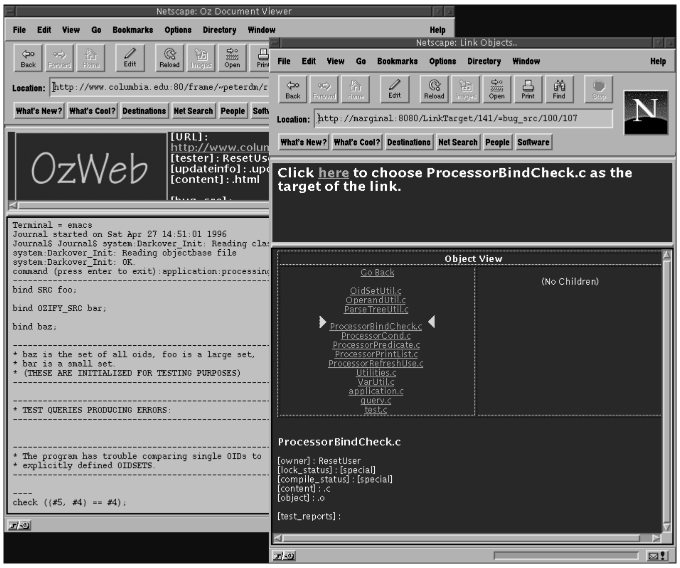

Figure 4: Linking QA Report and Code

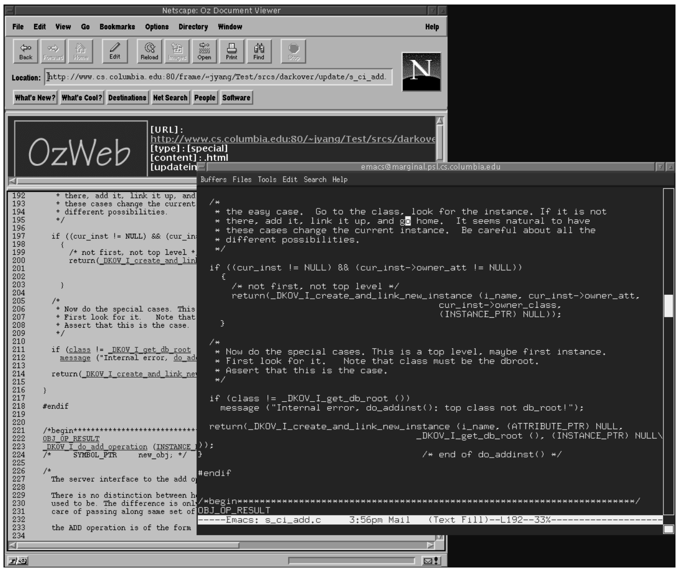

Figure 5: Cross-Linked Code and Tool

works slightly differently than for access through standard browser windows, since the **OzWeb** server transforms the URL and uses HTTP Redirect to force the browser to re-request the entity from its proper location. Subweb lookup is not needed, since GUI links always refer to subweb materials.

Revisit Scenario

We have developed an **OzWeb** environment supporting our motivating scenario. Users edit, compile, test, etc. C code for the evolving OODB system and query processor stored in the subweb. Source code is automatically converted to fully cross-referenced hypertext via process automation that invokes a home-grown utility called Hi-C. The subweb is populated with the code, design, testing and informal materials described previously, which are divided among three Columbia websites.

Laura first views some test reports regarding the baseline code through a standard Web browser. Then she requests the **OzWeb** GUI and adds an external link between one test report and a code file that she believes likely to contain the bug, as depicted in Figure 4. A *link* operation in the overloaded rule effect automatically adds a reverse link. In Figure 5, Laura takes a look at the hypertext source code in her browser, and selects the `edit` task link from the Action Menu. A regular ASCII editor tool is invoked (and the HTML is generated), but an alternative process could support a Web editor and strip off the HTML tags before presenting to the C compiler. The editor is forked by the subweb proxy and displays on Laura's workstation. The `edit` rule indicates to use the plain source code as the tool script argument, not the hypertext version. After the `edit` task completes, Hi-C incrementally updates the cross-references.

Meanwhile, Hugh searches through informal "proprietary" documents like email, newsgroup archives, etc. As shown in Figure 6, he uses **OzWeb** *search* to find subweb documents containing the string `''Darkover_GetNameByOID''`, because he is working on this function. Then Hugh decides to search on WWW, not just the subweb, using a standard search engine. He finds a "public" page he deems relevant, and adds it to the subweb as shown in Figure 7.

Evaluation

The most significant limitation of **OzWeb** 0.2 is we ignore version and configuration management. A process designer may explicitly include version control in the process, as currently in Oz, but we leave investigation of built-in Web versioning schemes to others [20].

The implementation presented here is **OzWeb** 0.2. We previously developed another system called **OzWeb** (effectively version 0.1), described in [9]. It provided a Web-based GUI to Oz similar to some commercial workflow systems. The original architecture involved an HTTP proxy server and would work with any HTTP browser, but that is where the similarity ends. There was no hypercode: none of the entities were in HTML format or included embedded links. There were no subwebs, and thus no ability to incorporate external materials from the Web or elsewhere: all documents were resident in the system's objectbase. The browser had to explicitly request Oz functionality via a URL of the form "http://oz/command/argument1/argument2/..."; if the "oz" site was not included in the URL, nothing happened beyond the usual retrieval of the page from its

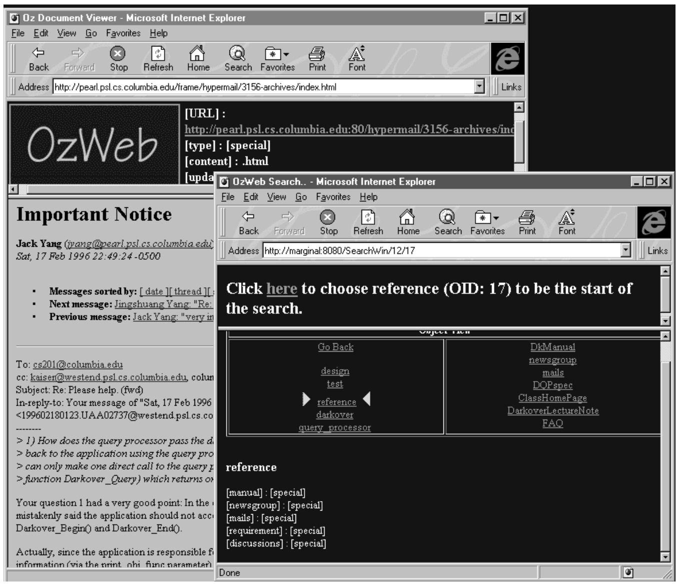

Figure 6: Subweb-specific Search

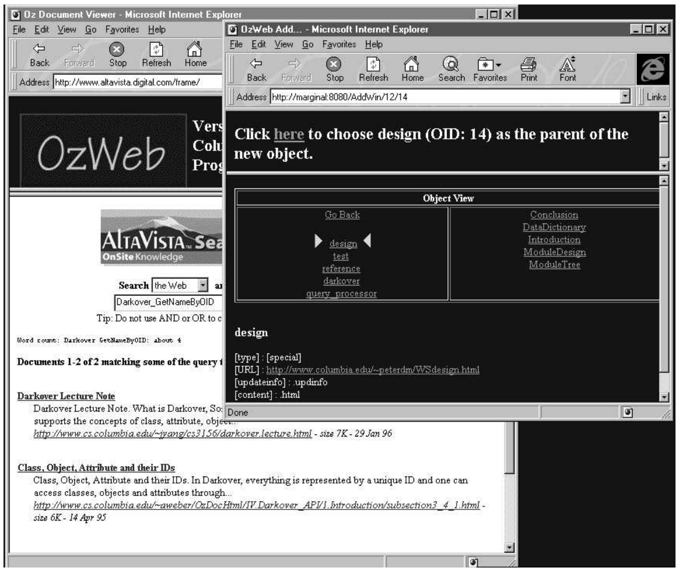

Figure 7: Adding to a Subweb

website. This is an important distinction: our subweb proxy supports automatic application of full groupspace services (process, transactions, tool invocation, etc.) to *any* URL represented in the corresponding subweb, whose Web page may reside anywhere on the Internet.

The functionality (beyond Oz itself) was originally implemented in a heavyweight proxy server, a modified Oz tty client. It used HTTP to communicate with Web browsers, but directly used the peculiar Oz client/server protocol to communicate with the Oz server. In contrast, the main functionality in 0.2 resides in the **OzWeb** server, which communicates with the proxy using HTTP (and will later communicate with other system components, such as tool servers, via an Oz -specific protocol on top, dubbed OzHTTP). The **OzWeb** server was implemented by extensive modifications to the old Oz server. The 0.2 proxy is very lightweight, the main thing it does differently from a standard HTTP proxy (e.g., used for caching purposes) is to query the **OzWeb** server to see whether or not the requested URL is in the subweb and (optionally) add on presentation of subweb object attributes; otherwise it just does everything the browsers and **OzWeb** server tell it to do via standard HTTP.

We also developed the experimental subweb server described in [22]. We used a rather convoluted implementation involving CGI rather than a proxy. It was thus impossible to intercept *all* Web accesses as in our current system. No groupspace services were supported, it was purely an organizational facility. We explored "view objects", to describe process-specific reformatting of HTML materials, not yet implemented in **OzWeb**.

There are several limitations of **OzWeb** 0.2 regarding our goals for subwebs and groupspaces. For example, each subweb proxy is currently hardwired to a particular subweb. For an end-user to move to a different groupspace, he/she must designate a different proxy in his/her Web browser configuration. It is not possible for an end-user to be operating in multiple groupspaces at the same time from the same browser. This limitation is easy to remove by connecting each subweb proxy to a directory service such as Lightweight Directory Access Protocol (LDAP [23]) through which it can interact with any number of registered subwebs. We did not change in the implementation yet, because of a conceptual complication regarding what should be done when a browser accesses a URL represented in two or more subwebs; this will be part of our future research on extending Oz alliances to **OzWeb**.

Tool scripts are spawned by the subweb proxy, and assume X Windows to redirect the tool's GUI to the user's screen. Pure command line tools also work. The scripts and thus tools are run under the operating system userid under which the subweb proxy is invoked, say "oz"; there may be access control, security, etc. reasons why it would be better if tools were executed under the userids of the user controlling that tool instance. And there may be performance reasons for launching tools on the user's own hosts or at least on some machine(s) other than where the subweb proxy is executing. The former could be achieved by installing the subweb proxy with "root" privileges, but the remote end-user would be required to have a userid on the host where the subweb proxy runs (and there may be conflicts between userids from different administrative domains, e.g., multiple users of the same groupspace with userid "yang").

We took a different approach in **OzWeb** 0.1, where the browser launched tools via new MIME (Multipurpose Internet Mail Extensions) types and helper applications. All tools ran under the end-user's userid, and consumed computational resources on that user's workstation. But all relevant tools must then actually be *available* from the end-user's machine — introducing complications with respect to platform-compatibility, machine resource requirements, software licensing, etc. And it is difficult for multiple users to "share" a tool instance, directing their tasks to the same running tool instance serially or concurrently [19]. We are investigating a third model, where separate tool servers are introduced as a new architectural element and groupspace service that combines the advantages of both approaches.

RELATED WORK

WebMake [12] supports hierarchical structuring of Web files, checkout for editing, and invocation of file-based tools like `make`. A special Web client hides details from users; otherwise WebMake uses standard facilities: CGI and MIME types. Data may be temporarily transmitted to another locale for tool execution (e.g., to compile for a particular architecture) using XMosaic remote control. WAIBA [16] produced a suite of utilities useful to software development teams (and others), such as collaborative annotations, search engines that find what's changed in categories of interest, what's changed in recently viewed pages, what's out there that's "similar" to what the user is currently viewing, displays link composition to a given depth, displays browsing history graphically, etc. Meteor [18] is a transactional workflow engine that submits workflow tasks *to* Web browsers using CGI, but does not support workflow *over* Web accesses.

Hyperform [21] is a hypertext database with an extensible object-oriented schema. It allows choice between checkout of individual attributes or entire objects (with dirty reads), and conventional transactions over multiple objects. An application must be implemented by a method of the relevant class in the schema to use transactions, but methods written in Scheme can be added dynamically. DHT [13] overlays hypertext to add on transparent access and external organization to distributed, heterogeneous, autonomously-maintained repositories. Repository data is transformed to/from a common structured hypermedia format by a repository-specific gateway. Only navigational access is supported. Update requests are turned down if the object has changed since last retrieved.

WebCard [5] supports representation of Web pages in the style of email/news folders. Lightweight Databases [8] extends HTML to map relational database schemas onto hypertext documents, to support queries that rely on semantic knowledge of the structure and content of documents. Relationships among hypertext pages augment the usual links. However, data content must be modified to include the entity class, attributes, and relationships. Hyper-G [2] supports hierarchical structuring of aggregate collections of Web pages, with collections spread across websites. Links are represented externally to the hypermedia content. Scaling is supported by replication and caching (with weak consistency). Hyper-G uses its own format, but converts to HTML when serving a WWW client.

Ockerbloom [15] proposes an alternative to MIME types, called Typed Object Model , that could be employed underneath our subwebs. Object types exported from anywhere on the Internet can be registered in "type oracles", specialized servers which may communicate among themselves to uncover the definitions of types registered elsewhere. Web clients who happen upon a type they do not understand can ask one of the type oracles how to convert it into a known supertype.

CONTRIBUTIONS AND FUTURE WORK

We have designed a general approach to hypercode environments centered on subweb repositories and groupspace services, investigated many of the technical issues in depth, developed a feasible architecture and implementation technique based on WWW technology, and realized a prototype environment framework and sample environment in **OzWeb**.

OzWeb 0.2 reuses Oz's with its process engine and transaction manager more-or-less "as is"; the same OODB is used in the subweb implementation. In future work we would like to exploit our recent componentization direction, where these components have been tugged apart, are in principle replaceable within Oz, and have been experimentally integrated with foreign systems. For example, a later version of **OzWeb** might employ an alternative process engine.

Our architecture seems suited to adapting existing client/server process-centered environments to a groupspace/subweb system. The server already provides basic groupspace functionality and would be augmented by the subweb mechanism, as we described for

Oz. Clients are replaced by Web browsers and proxy servers. Peer/peer environments where the peers already share a common repository [17] may be adaptable to multiple groupspace servers sharing a common subweb server, but investigation is outside our scope.

It is tempting to consider how some other distributed computing infrastructure, other than the Web, might serve as the basis for subweb and groupspace implementation. We have looked at CORBA [14] as a candidate, but it is currently lacking the key ingredient: a standard component that supports complete interception and mediation in the style of HTTP proxy servers. This might be an appropriate avenue to explore for future distributed computing standards.

ACKNOWLEDGEMENTS

We thank George Heineman, Dick Taylor, Sankar Virdhagriswaran and Alex Wolf for useful technical discussions. Laura Xiaoyu Xu, Hugh Chen Zhang, and other COMS 3156 students participated in our scenario.

The Programming Systems Lab is funded in part by DARPA monitored by Air Force Rome Lab F30602-94-C-0197 and in part by NSF CCR-9301092. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US government, DARPA, Air Force or NSF.

REFERENCES

[1] K. M. Anderson, R. N. Taylor, and E. J. Whitehead, Jr. Chimera: Hypertext for heterogeneous software environments. In *1994 European Conference on Hypermedia Technology*, pages 94–107, September 1994.

[2] K. Andrews, F. Kappe, and H. Maurer. Serving information to the Web with Hyper-G. In *3rd Intl. World-Wide Web Conference*, April 1995.

[3] I. Ben-Shaul and G. E. Kaiser. *A Paradigm for Decentralized Process Modeling*. Kluwer, 1995.

[4] C. Brooks, M. S. Mazer, S. Meeks, and J. Miller. Application-specific proxy servers as HTTP stream transducers. In *4th Intl. World Wide Web Conference*, pages 539–548, December 1995.

[5] M. H. Brown. WebCard: Integrated and uniform access to mail, news, and the Web. Technical Report 139a, DEC Systems Research Center, July 1996.

[6] U. Dayal, H. Garcia-Molina, M. Hsu, B. Kao, and M.-C. Shan. Third generation TP monitors: A database challenge. In *1993 SIGMOD Intl. Conference on Management of Data*, pages 393–398, May 1993.

[7] Defense Advanced Research Projects Agency. *Evolutionary Design of Complex Systems*, July 1996. http://www.ito.darpa.mil/ResearchAreas/EDCS.html.

[8] S. Dobson and V. Burrill. Lightweight databases. In *3rd Intl. World-Wide Web Conference*, April 1995.

[9] S. E. Dossick and G. E. Kaiser. WWW access to legacy client/server applications. In *5th Intl. World Wide Web Conference*, pages 931–940, May 1996.

[10] A. K. Elmagarmid (ed). *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[11] G. E. Kaiser. Cooperative transactions for multi-user environments. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter 20, pages 409–433. ACM Press, 1994.

[12] Michael Baentsch, Georg Molter and Peter Sturm. WebMake: Integrating distributed software development in a structure-enhanced Web. In *3rd Intl. World-Wide Web Conference*, April 1995.

[13] J. Noll and W. Scacchi. A hypertext system for integrating heterogeneous autonomous software repositories. In *3rd Irvine Software Symposium*, pages 49–60, April 1994.

[14] Object Management Group. *The Common Object Request Broker: Architecture Specification Revision 2.0*, July 1995. www.omg.org/corba2/cover.htm.

[15] J. Ockerbloom. Introducing structured data types into Internet-scale information systems, May 1994. Carnegie Mellon University School of Computer Science PhD Thesis Proposal. www.cs.cmu.edu/afs/cs.cmu.edu/-user/spok/www/proposal.html.

[16] OSF Research Institute. *Intelligent Browsing Assistant for the World Wide Web and GroupWare for the Web*, October 1995. www.osf.org/www/waiba/index.html.

[17] B. Peuschel and S. Wolf. Architectural support for distributed process centered software development environments. In W. Schäfer, editor, *8th Intl. Software Process Workshop*, pages 126–128, March 1993.

[18] A. Sheth. Private communication, June 1996. See lsdis.cs.uga.edu/workflow/.

[19] G. Valetto and G. E. Kaiser. Enveloping sophisticated tools into process-centered environments. *Journal of Automated Software Engineering*, 1996. In press.

[20] J. Whitehead. Working group on versioning and configuration management of World Wide Web content, June 1996. www.ics.uci.edu/~ejw/versioning/.

[21] U. K. Wiil. Hyperform: Rapid prototyping of hypermedia services. *Communications of the ACM*, 38(8):109–111, August 1995.

[22] J. J. Yang and G. E. Kaiser. An architecture for integrating OODBs with WWW. In *5th Intl. World Wide Web Conference*, pages 1243–1254, May 1996.

[23] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol, March 1995. Network Working Group Request For Comments: 1777, andrew2.andrew.cmu.edu/rfc/rfc1777.html.