A Configuration Process for a Distributed Software Development Environment (Extended Abstract)

Israel Z. Ben-Shaul
Gail E. Kaiser

Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
tel: 212-939-7085/fax: 212-666-0140
israel@cs.columbia.edu

CUCS-022-93
August 18, 1993

Abstract

This paper describes work-in-progress on a configuration facility for a multi-site software development environment. The environment supports collaboration among geographically-dispersed teams of software developers. Addition and deletion of local subenvironment sites to a global environment is performed interactively inside any one of the existing local subenvironments, with the same user interface normally employed for invoking software development tools. This registration process is defined and executed using the same notation and mechanisms, respectively, as for the software development process. Each remote site is represented by a root object in the distributed objectbase containing the software under development; each local subobjectbase can be displayed and queried at any site, but only its root is physically copied at every site. Everything described in this paper has been implemented and is working, but since we are in the midst of experimentation, we do not expect that the "final" system will be exactly as described here.

1 Introduction

Large-scale software development projects are not always confined to a single site, or even to a single organization. A project may span multiple teams that are geographically dispersed across a wide area network such as the Internet. (We use the term "site" to mean an administratively cohesive Internet domain sharing a single network file system name space, e.g., cs.columbia.edu, as opposed to either a single host such as lafayette.cs.columbia.edu or a campus backbone such as columbia.edu.)

There is a spectrum of approaches to software development environment (SDE) support for such projects. At one end, each team chooses its own SDE and there is little concern with whether the local environments are compatible with each other. Or the teams might agree to choose the same *single-site* SDE, to remove the data conversion problem and provide a common vocabulary, but they still run distinct instances of the SDE and all sharing and collaboration across teams is done outside the environment. At the other end of the spectrum, each team shares the same instance of a *multi-site* SDE, which provides facilities for sharing and collaboration inside the environment.

This end of the spectrum is analogous to a distributed database system, while the other end is comparable to a set of independent databases. The database community has delineated a practical intermediate point, often termed federated databases, which permits a high degree of site autonomy [14]. The autonomy is usually with respect to one or both of two criteria, schema and system: the sites may employ the identical system, but devise their own schema independently (also known as a homogeneous federation), and/or they may select different database systems from among those supported by the federation "glue" (heterogeneous).

We have adapted the former criteria to SDEs, specifically a subclass of SDEs known as *process-centered environments* (PCEs). A PCE is a generic kernel that is parameterized by a *process model*, which defines the software development process for that instance of the environment; the kernel assists the users in carrying out the defined process, by guiding them from one step to another, enforcing the constraints and implications of process steps, and/or automating portions of the process [15, 17]. A geographically distributed PCE would permit each site to specify its own process model, including the desired collaboration with other sites. We do not consider the latter criteria above, where distinct environment systems might inter

Our approach to multi-site process definition and execution [5] is outside the scope of this workshop. The focus of this paper is on the configuration of a global environment, and its reconfiguration over time while the long-term software development project is in progress. However, we exploit the process-centered aspect of the environment by defining the *registration process* in the notation normally used to specify a software development process, and executing this process using the engine normally used to invoke software development tools and maintain consistency among software artifacts. The registration process represents each site as an object in the object-oriented database containing the software artifacts, and the tool enveloping facilities are used to implement the communication for global configuration. (As with the software process, we could permit site-specific variability in the configuration process, but it would necessarily be limited to a common subschema for the site objects.)

We first describe the Marvel 3.1 system, a single-site PCE from which we are borrowing many of the process concepts (and as much code as possible!). Then we give an overview of a multi-site PCE, which we call Oz 0.2, focusing on the aspects of its operation relevant to configuration. (The current version of Oz is only *logically* multi-site, since we assume a common file system name

```
compile [?c:CFILE]:
    (and (exists PROJECT ?p  suchthat (ancestor [?p ?c]))
         (forall INC     ?i  suchthat (member   [?p.include  ?i]))
 (forall HFILE   ?h  suchthat (member   [?i.hfiles   ?h]))):

    # 0. If an INC object is archived, don't automatically chain here
    # 1a. If this CFILE has been analyzed or
    # 1b. had a previous compilation error.
    # -------------------------------------------------------------
    (and no_forward (?i.archive_status = Archived)            # 0.
         (or (?c.status = Analyzed)                           # 1a.
             no_chain (?c.status = ErrorCompile)))            # 1b.

    \{ BUILD compile ?c.contents ?c.object_code ?h.contents ?c.history "-g" \}

    # 0.a Chain to archive the module in which it is contained.
    # 0.b Update the object code time stamp
    # 1.  Update status of object.  Chain to dirty rules
    # -------------------------------------------------------
    (and (?c.status = Compiled)                         # 0.a
 (?c.object_time_stamp = CurrentTime));            # 0.b
    (?c.status = ErrorCompile);                         # 1.
```

Figure 1: CFILE Compile Rule From C/Marvel

space and ignore authorization and security issues.) We present the configuration process model module and explain how it works. The main contribution of this paper is that the (re)configuration process is fully integrated, meaning it can be treated by the process-centered environment exactly like any other process that one undertakes during software development — and it can be modified for existing and new environments using the same process evolution capabilities.

2 Marvel Background

The goal of the Marvel project was to develop a kernel for process-centered environments that guide and assist a team of users working on a medium-scale software development effort. The generic kernel must be tailored by an *administrator* who provides the schema, process model, coordination model and tool envelopes for a specific project. The schema classes define an objectbase containing the software system under development. Multiple inheritance, attributes of primitive, enumerated, binary file and text file types, aggregate composite objects and non-hierarchical links between objects can be declared. The X11 windows user interface supports graphical browsing and ad hoc queries; there is also a command line interface for terminals and batch scripts.

The process (or workflow) is given in a process modeling language [13]. Each process step is encapsulated in a *rule* with a name and typed parameters. The body of a rule consists of a query to bind local variables (e.g., included ".h" files); a complex logical condition on the actual parameters and bound variables, which must be satisfied prior to complete the step; an optional activity; and a set of effects that each assert one of the activity's possible results (only one effect if there is no activity). Forward and backward chaining over the rules enforces consistency and automates tool invocations. Enforcement and automation are two forms of "enaction", the term used in the community for any computer-aided support for process. A sample rule for compiling (cc) a C file is shown in Figure 1, Process enaction is mainly user-driven, as opposed to system-driven. The user chooses when to try a particular process step, and then Marvel selects the "closest" matching rules (there may be more than one) and fires each of these rules in turn until it finds one whose condition is already satisfied or can be satisfied by backward chaining [3]. The activity, if any, of this rule is then executed. Afterwards, one of the effects is selected according to a status code returned by the activity, and Marvel forward chains to any other rules that are implications of this effect. If none of the conditions can be satisfied, however, then it is not possible to undertake that process step at this time.

Note that since rules have multiple effects, it may be possible that an attempted backward chain results in an undesired effect, but the chain is not then "undone" because that would be counterproductive (consider a rule that compiles source code with the intent to generate correct object code, and instead produces syntax error messages); nevertheless, in the case where there are multiple rules that might produce the desired effect, another one is attempted until the original condition is satisfied or all alternatives are exhausted.

The condition of the rule shown in Figure 1 permits backward chaining from it to `archive` all the include directories of the project (predicate 0), but forward chaining into it from `archive` is prevented by the "no_forward" directive. Forward chaining into this rule is permitted after linting a C file, which becomes the parameter (predicate 1a). Chaining is prohibited into or out of the `compile` rule from another instance of the `compile` rule, due to the "no_chain" directive (there is also a "no_backward" directive).

Additional details about the rule formalism and its chaining engine are given in [4, 11, 10].

Conventional file-oriented tools are integrated into a Marvel process without source modifications, or even recompilation, through an *enveloping* language [7]. The rule activity indicates the tool and envelope name, with input literals and attributes to be supplied as arguments and output variables for binding to returned results; an implicit status code selects the actual effect from among those given in the rule. The body of an envelope is a shell script, written in any one of the conventional Unix shell languages: ksh, csh, or sh. Existing software can be *immigrated* from the file system into a Marvel objectbase using the Marvelizer utility [16].

Multiple users are supported by a client/server architecture [6]. A client provides the user interface, checks the arguments of commands, and forks tool envelopes, while the process engine, synchronization management and objectbase reside in the central Marvel server. Scheduling is FCFS, with rule chains interleaved at the natural breaks provided when clients execute activities. Clients may run on the same or different hosts as the server, but the enveloping facility assumes a shared network file system. The external view is illustrated in Figure 2.

Additional details about multi-user issues, primarily concurrency control, are given in [2, 1, 9]. Marvel's support for schema and process evolution while a long-term project is in progress is described in [12].

We completed Marvel 3.0, the first multi-user version, in Fall 1991. 3.0.1 was developed using the C/Marvel environment on top of Marvel 3.0, and released in Spring 1992. The "final" Marvel 3.1 was released in March 1993, and consists of about 154,000 lines of C, yacc and lex code. 3.1 supports a choice of three user interfaces — XView, Xlib and tty — and runs on SparcStations with SunOS 4.1.3, DecStations with Ultrix 4.3, and IBM RS6000s with AIX 3.2. Our main example environments are included as part of the distribution: C/Marvel, for Marvel's own development and testing; P/Marvel, for installation and evolution of environment definitions; and our solution to

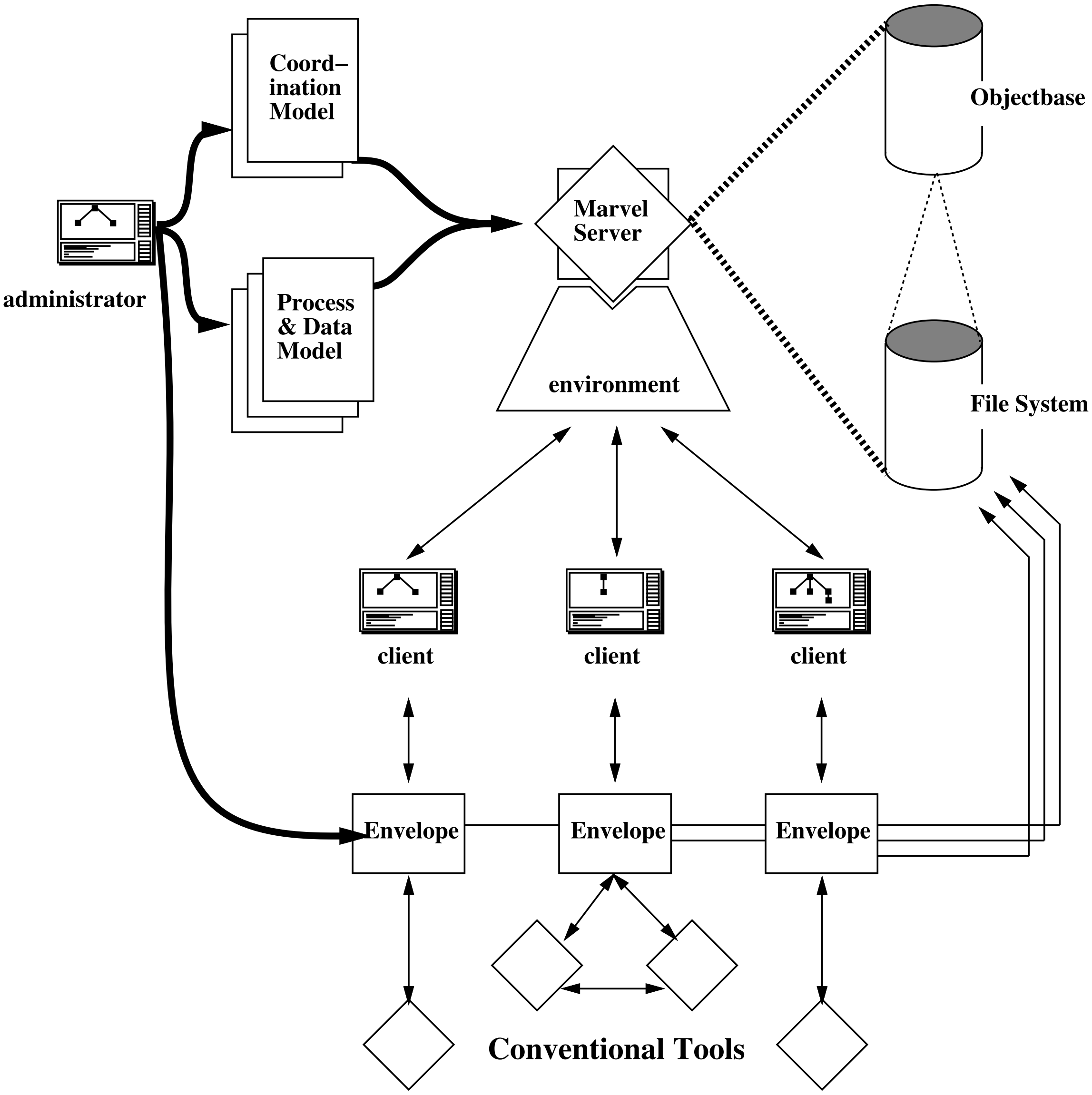

Figure 2: Generic Marvel 3.x Environment

the benchmark problem developed for the 7th International Software Process Workshop [8]. Marvel has been licensed to about 50 institutions to date; licensing is restricted to educational institutions and funding sponsors.

3 Oz Overview

An Oz global environment is depicted in Figure 3. Three local subenvironments are shown; they might reside at three different Internet sites, all at the same site, or a combination. In the sequel, we will use the terms site and subenvironment interchangeably, but it should be understood that multiple distinct subenvironments of the same global environment can be co-located at the same Internet site. (And of course subenvironments of unrelated global environments can also be co-located.)

The objectbase containing the software artifacts under development is distributed across the three sites, i.e., each subenvironment has its own subobjectbase. File attributes in the objectbase actually contain only file pathnames as their values, corresponding to a directory hierarchy in the normal file system — where the contents reside. The relevant portions of the file system are intended to be accessible to tools only through the environment, however.

There is no replication of either objectbase or file system, except for the root objects as described below. The distribution is not transparent; instead, each subobjectbase is intended to contain only those artifacts produced by its own subenvironment. However, any subobjectbase can be displayed for browsing purposes at any site, in which case only a display image of the subobjectbase *structure* is transferred and cached locally rather than the full *contents* of the subobjectbase. By structure, we mean the names, types and relationships among objects, but not the values of their attributes (which do not reflect relationships to other objects). Further, any subobjectbase can be searched for objects according to navigational and/or associative queries, either ad hoc or embedded in a process step, in which case the matched objects are temporarily copied to the querying site.

The figure illustrates the leftmost subenvironment as currently dormant. However, if a client at one of the other two sites wishes to browse or query the subobjectbase stored there, then the client would communicate with that site's Connection server, to automatically bring up an instance of the Oz server on the dormant subenvironment's default host. (A client need not reside on the same host as its local servers, only at the same Internet site sharing a network file system.) From then on the client would communicate directly with the remote Oz server, in addition to the client's local Oz server and any other remote Oz servers previously connected but not yet disconnected. Other local and remote clients requesting to access the same previously dormant subobjectbase would connect to this same Oz server. After its last client disconnects, an Oz server automatically shuts down; in contrast, a Connection server runs continuously. (Connection servers versus Oz servers should not be confused; when we say just "server", it means an instance of the Oz server.)

Every subenvironment participating in a global environment is represented by a distinct root object; the objectbase is thus a forest, not a strict tree. (There are also links among objects, in addition to composition relationships, so the objectbase is actually a general directed graph; this distinction is not germane to the discussion.) These root objects are always part of the displayed image at all clients at all sites of the global environment.

When a client is not connected to a particular remote server, the root object corresponding to that remote subobjectbase is effectively a "stub" containing only partial information. This includes

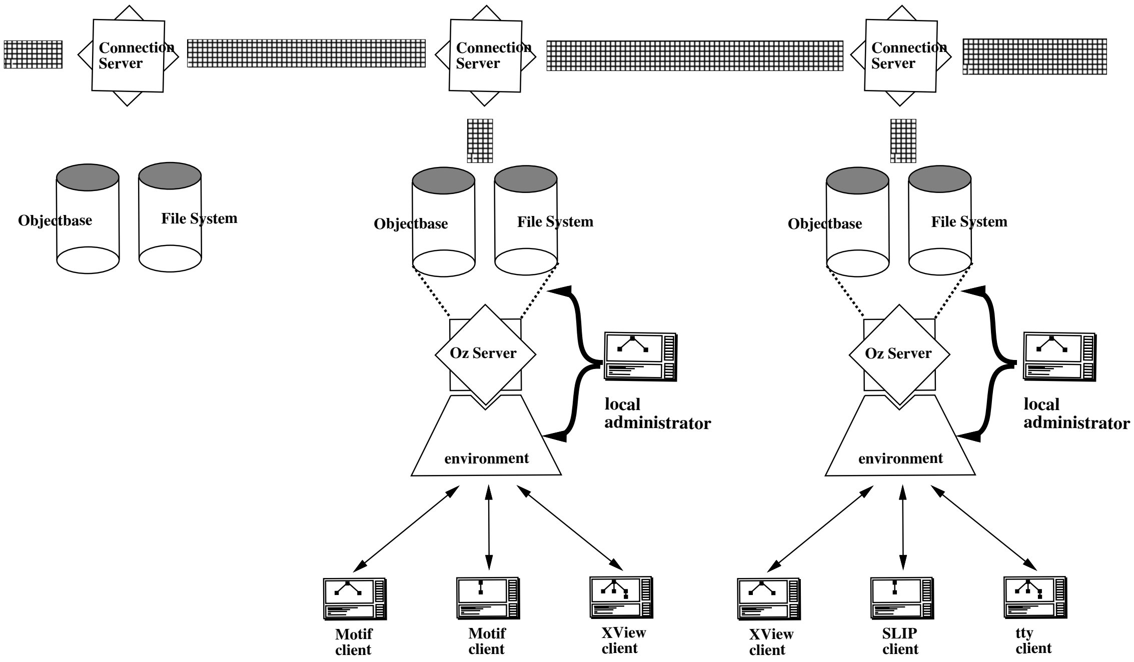

Figure 3: Generic Oz Global Environment

static information supplied during registration (described in the next section), which is always correct thereafter, and *dynamic* information left over from the last connection, which may or may not be up-to-date. The static information is used to identify the remote site for a given global environment and local subenvironment. The remote connection server consults its own site-specific internal tables to determine the dynamic information needed to set up a connection.

A root object is filled in with the current values of the dynamic attributes when a connection is established, so there is no need for further recourse to the Connection server while the connection remains up. A distinguished attribute represents a hook to the structural image of the composition of the remote subobjectbase. The objects shown in this structure can be selected using the same interface as for the local subobjectbase; the only noticeable difference in responding to a mouse click to present the contents of a local versus remote object would be due to network delay. An example screen dump, with a local subobjectbase, an open remote subobjectbase and two closed ones, is shown in Figure 4.

4 Implementation of Configuration Facilities

The `open-remote` and `close-remote` commands above are implemented as *built-in commands*, meaning they are hard-wired into the Oz kernel and therefore available in every Oz environment (like `help`, `quit` and `refresh`). However, the vast majority of commands in any environment represent process steps, or rules, that are available only in an environment tailored by that process model.

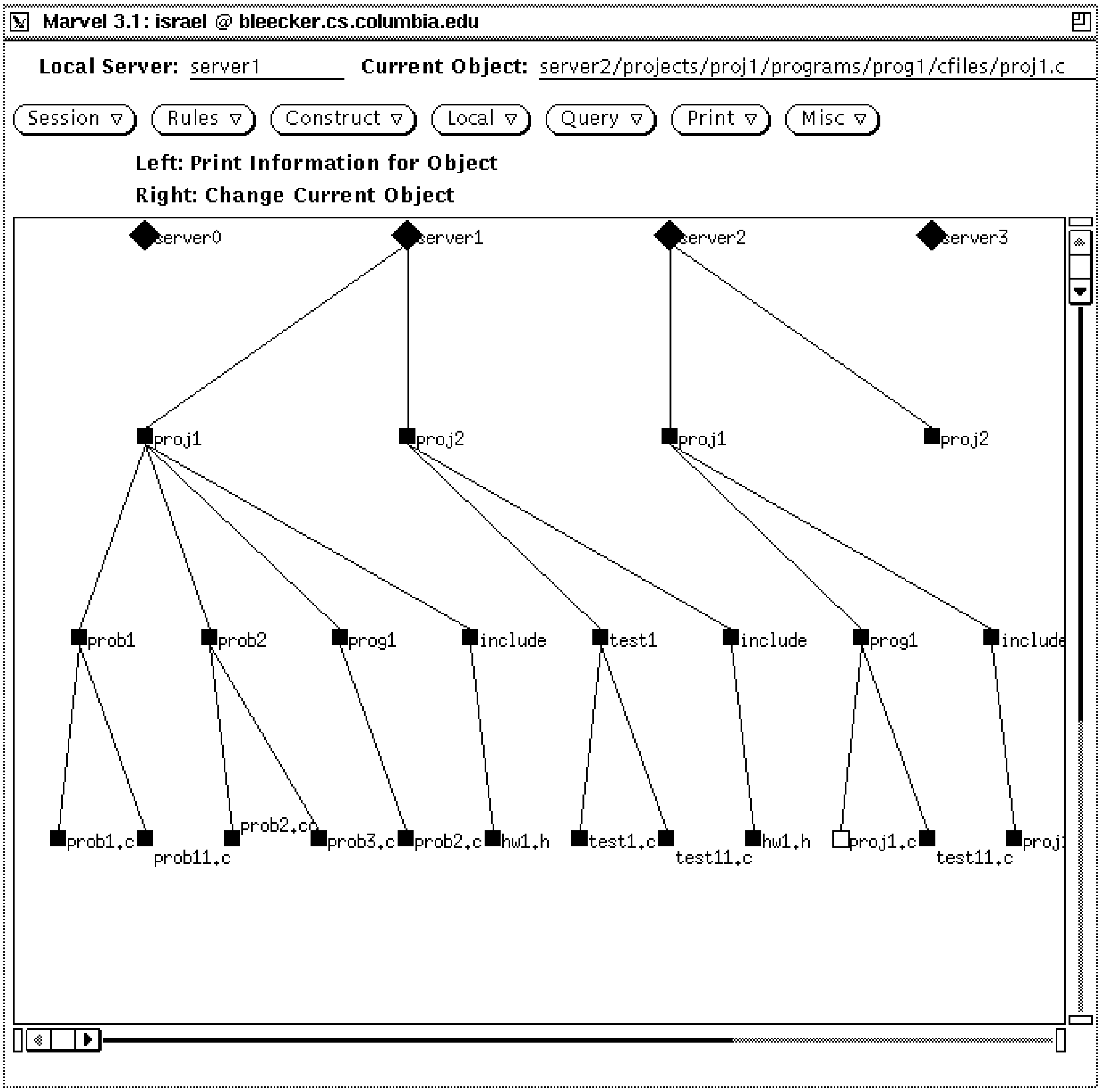

Figure 4: Oz Screen Dump

```
# GROUP is the top-level class, a root in the objectbase forest.

GROUP :: superclass ENTITY;

# Static Information
   env_name     : string;           # logical name
   env_id       : integer;
   subenv_name  : string;           # initially directory path name
   subenv_id    : integer;
   site_name    : string;           # e.g.: cs.columbia.edu
   site_ip_addr : string;           # dotted format, e.g.: 128.59.16.20
# Dynamic information
   active_host  : string;           # e.g.: lafayette.cs.columbia.edu
   host_ip_addr : string;           # dotted format, e.g. 128.59.24.51
   port         : integer = 0;      # port number, if active
   local        : boolean;          # TRUE if local, FALSE if remote
   state        : (New, Initialized, Defunct ) = New;
   active       : boolean = false;  # TRUE if active, not guaranteed
   build_status : (Built, NotBuilt, Initialized) = Initialized;
   projects     : set_of PROJECT;   # for compatibility with Marvel
   subenv_ob    : set_of ENTITY;    # The objectbase resides here.
end
```

Figure 5: Class for Root Objects

Figure 5 shows the `GROUP` class. By definition, every instance of the `GROUP` class in any Oz environment is a root object of the environment's objectbase. The static information constitutes the "stub" mentioned above; each stub represents a single site and a copy of the stub resides in every local subenvironment. The dynamic information in a particular copy is updated whenever a local client opens a connection to the site. Thus we refer to the object as a copy rather than a replica, since the copies in different subenvironments may diverge depending on the recency of their last connections.

Figure 6 gives the rule, i.e., the process step, for adding a new site to a multi-site environment. The `register_subenv` rule may be fired from *any* site already participating in the relevant environment. The server binds the root object representing the pre-existing local subenvironment to `lse` and all the root objects representing pre-existing remote subenvironments to `se`, and then the client forks the `register_subenv` tool envelope.

This "tool" prompts the user for the new site's static information, including its location, and copies the corresponding instance of `GROUP` into the subobjectbase for every existing subenvironment. As written, the envelope performs no error checking, so there is only one (rather meaningless) effect. A more mature registration process might incorporate a tool envelope that detects the occurrence of common problems and selects a distinct effect for each, perhaps to forward chain into a semi-automated "exception handler".

Further, the `register` rule should properly execute as a distributed transaction, so if two or more users attempt to add the same site concurrently, all except one would automatically be aborted and thus rolled back. We are developing an extended transaction facility suitable for long duration, interactive, and cooperative applications [9], which at present is implemented only with respect to a single server (within a single site). In the meantime, concurrency control and other failures affecting multiple sites must be recovered manually.

Figure 7 shows the `send_subenv_map` rule, which initializes the new subenvironment by creating root

```
# collect information about the new subenv,
# and replicate it in all current subenvs using a batch client
# consisting of adding the object and calling init_remote_subenv

register_subenv [?new_name:LITERAL]:

    # collect all remote subenv objects
   (and
    (forall GROUP ?se suchthat (?se.local = false))
    (exists GROUP ?lse suchthat (?lse.local = true)))
  :

# this envelope actually does the replication in remote subenvs
    { REGISTER register_subenv ?new_name ?se.subenv_name ?se.subenv_id
               ?lse.subenv_name ?lse.subenv_id}
    # a check should be made here.
    no_assertion;


# called from within an envelope (in batch mode), from all
# remote subenvs, to assign the proper values to the object which
# was just added by register_subenv

hide init_remote_subenv [?env_name:LITERAL, ?env_id:LITERAL,
            ?subenv_id:LITERAL, ?subenv_name:LITERAL,
    ?site_name:LITERAL, ?site_ip_addr:LITERAL]:

   (exists GROUP ?new suchthat (?new.state = New)):

    {}

    (and (?new.env_name = ?env_name)
         (?new.env_id  =  ?env_id)
         (?new.subenv_id  = ?subenv_id)
         (?new.subenv_name  = ?subenv_name)
         (?new.site_name = ?site_name)
         (?new.site_ip_addr  = ?site_ip_addr)
         (?new.local = false));
```

Figure 6: Add New Site

```
# initialize the newly created subenv, by sending to it the
# environment map.

send_subenv_map [?nse:GROUP]:

    # collect all subenv objects except the new one
    (forall GROUP ?s suchthat (?s.subenv_name <> ?nse.subenv_name))
    :
    no_chain(?nse.state = New)

    # this envelope actually copies the map to the new subenv
    {
       REGISTER send_subenv_map
                 # new subenv's identification and location
                 ?nse.Name ?nse.env_name ?nse.env_id
                 ?nse.subenv_id ?nse.subenv_name
                 ?nse.site_name ?nse.site_ip_addr
                 # all information of all subenvs.
                 ?s.Name ?s.env_name ?s.env_id  ?s.subenv_id ?s.subenv_name
                 ?s.site_name ?s.site_ip_addr
    }

    (?nse.state = Initialized);


# called from within an envelope (by invoking a batch client)
# of send_subenv_map, this assigns to all objects of the environment
# map the proper values.

hide init_subenv_map [?new:GROUP, ?env_name:LITERAL, ?env_id:LITERAL,
            ?subenv_id:LITERAL, ?subenv_name:LITERAL,
    ?site_name:LITERAL, ?site_ip_addr:LITERAL, ?local:LITERAL]:
    :
    {}

    (and (?new.env_name = ?env_name)
         (?new.env_id  =  ?env_id)
         (?new.subenv_id  = ?subenv_id)
         (?new.subenv_name  = ?subenv_name)
         (?new.site_name = ?site_name)
         (?new.site_ip_addr  = ?site_ip_addr)
         (?new.local = ?local)
         (?new.state = Initialized));
```

Figure 7: Initialize Local Information About Global Environment

```
# Remove a subenv, by removing the object from all remote subenvs
# and removing all subenvs from the subenv which is removed.
# leave the subenv "disconnected", but don't remove
# it, so it can  be  at a later point.
#
# In order to be able to delete root objects, we use a hack:
# bind to symbol empty no objects, and use it as the second
# operand to the delete, which will allow me to delete top level
# objects.

deregister_subenv [?lse:GROUP]:

    # collect all remote subenv objects
    (and
    (forall GROUP ?se suchthat (?se.local = false))
    (forall GROUP ?empty suchthat (?empty.Name = bogus)))
   :
    no_chain (?lse.local = true)

# this envelope removes the subenv object from remote subenvs
    { REGISTER deregister_subenv ?se.Name ?se.subenv_name ?lse.Name}
    (delete [?se ?empty] );
```

Figure 8: Delete Existing Site

objects in its subobjectbase — one for each of the other sites. In a mature registration process, it might be desirable for this process step to always be executed as part of the same process fragment, i.e., rule chain, as the `register_subenv` process step.

The `deregister_subenv` rule in Figure 8 is used to remove a site from the global environment. It deletes the corresponding root object from all the other subenvironments, and it deletes the root objects representing these other subenvironments in the site's own subobjectbase. The subenvironment itself is only split off from the global environment, but it is not destroyed; the former subenvironment can continue operation on its own as a single-site environment, and may be rejoined into this or another multi-site environment later.

Similarly, the `register_subenv` and `send_subenv_map` process steps apply both to joining an existing stand-alone local environment into a global environment as well as constructing an entirely new subenvironment from scratch. That is, the environment structure including its local subobjectbase is created automatically if it does not already exist. (One useful application of this feature is to upgrade an existing Marvel environment into an Oz single-site environment, and then join it into a multi-site environment.)

The "batch" facility mentioned in some of the comments (in the figures) refers to a recursive invocation of a new Oz client from within an envelope forked by an existing client. The new client connects to the same local server, performs the sequence of commands listed in a batch script, and terminates. This is the easiest way to add or delete a set of objects from an existing objectbase, although the built-in `add` and `delete` operations can also be invoked in the effects of rules and thus a sequence of adds and deletes can, in principle, be automated by the process enaction engine.

5 Conclusions

Everything described in this paper is implemented and working in Oz 0.2. Due to lack of time, we have concentrated in this extended abstract on *how* everything works. In the full paper, we will address in detail *why* we choose to do things this way, including the tradeoffs with respect to the spectra of alternatives, and briefly explain how one can modify the registration process simply by changing the rules (and tool envelopes). We will compare and contrast to related work, although most computer-supported cooperative work applications involve entirely external and/or ad hoc configuration mechanisms, and few consider geographical distribution. We will also supply listings of the configuration "tool" envelopes in an appendix, if desired by the reviewers — they total about 370 lines.)

Acknowledgments

Will Chou implemented a rudimentary client and connection server that do not assume a shared network file system, but this code has not yet been integrated into Oz . Zhongwei Tong is responsible for most of the Oz/Marvel process model, with which we parameterized Marvel 3.1 to assist us in tearing apart Marvel's own code to contribute to the Oz development effort.

References

[1] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University, February 1992. CUCS-001-92.

[2] Naser S. Barghouti. Supporting cooperation in the MARVEL process-centered SDE. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 21–31, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[3] Naser S. Barghouti and Gail E. Kaiser. Modeling concurrency in rule-based development environments. *IEEE Expert*, 5(6):15–27, December 1990.

[4] Naser S. Barghouti and Gail E. Kaiser. Scaling up rule-based development environments. *International Journal on Software Engineering & Knowledge Engineering*, 2(1):59–78, March 1992.

[5] Israel Z. Ben-Shaul. Oz: A decentralized process centered environment. Technical Report CUCS-011-93, Columbia University, Department of Computer Science, April 1993. PhD Thesis Proposal.

[6] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring 1993.

[7] Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 218–227, Redondo Beach CA, October 1991. IEEE Computer Society Press.

[8] Dennis Heimbigner and Marc Kellner. Software process example for ISPW-7, August 1991. /pub/cs/techreports/ISPW7/ispw7.ex.ps.Z available by anonymous ftp from ftp.cs.colorado.edu.

[9] George T. Heineman. A transaction manager component for cooperative transaction models. Technical Report CUCS-017-93, Columbia University Department of Computer Science, July 1993. PhD Thesis Proposal.

[10] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti, and Israel Z. Ben-Shaul. Rule chaining in MARVEL: Dynamic binding of parameters. *IEEE Expert*, 7(6):26–32, December 1992.

[11] Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler, and Robert W. Schwanke. Database support for knowledge-based engineering environments. *IEEE Expert*, 3(2):18–32, Summer 1988.

[12] Gail E. Kaiser, Israel Z. Ben-Shaul, George T. Heineman, and Wilfredo Marrero. Process evolution for the MARVEL environment. Technical Report CUCS-047-92, Columbia University Department of Computer Science, April 1993.

[13] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[14] Sudha Ram, editor. *Special Issue on Heterogeneous Distributed Database Systems*, volume 24:12 of *Computer*. IEEE Computer Society Press, December 1991.

[15] *2nd International Conference on the Software Process: Continuous Software Process Improvement*, Berlin, Germany, February 1993. IEEE Computer Society Press.

[16] Michael H. Sokolsky and Gail E. Kaiser. A framework for immigrating existing software into new software development environments. *Software Engineering Journal*, 6(6):435–453, November 1991.

[17] Ian Thomas, editor. *7th International Software Process Workshop: Communication and Coordination in the Software Process*, Yountville CA, October 1991. IEEE Computer Society Press.