Incremental Process Support for Code Reengineering: An Update (Experience Report)

Gail E. Kaiser George T. Heineman
Peter D. Skopp Jack J. Yang
Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
212-939-7081/fax:212-666-0140
kaiser@cs.columbia.edu

CUCS-007-96
15 February 1996

Abstract

Componentization is an important, emerging approach to software modernization whereby a stovepipe system is restructured into components that can be reused in other systems. More significantly from the system maintenance perspective, selected components in the original system can be completely replaced, e.g., the database or user interface. In some cases, a new architecture can be developed, for example to convert a monolithic system to the client/server paradigm, and the old components plugged into place along with some new ones. We update a 1994 publication in this conference series, where we proposed using process modeling and enactment technology to support both construction of systems from components and re-engineering of systems to permit component replacement. This paper describes our experience following that approach through two generations of component-oriented process models.

Keywords: Cross-Referencing, Object-Oriented Database, Process-Centered Environments, System Build, Tool Enveloping.

This paper is based on work sponsored in part by Advanced Research Project Agency under ARPA Order B128 monitored by Air Force Rome Lab F30602-94-C-0197, in part by National Science Foundation CCR-9301092, and in part by New York State Science and Technology Foundation Center for Advanced Technology in High Performance Computing and Communications in Healthcare NYSSTF-CAT-95013. Heineman is supported in part by an AT&T Fellowship. Skopp is now an employee of D.E. Shaw & Co. 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 or NYS government, ARPA, Air Force, NSF, NYSSTF, AT&T or D.E. Shaw & Co.

This page intentionally left blank.

1 Introduction

Componentization is an important, emerging approach to software modernization. Componentization, as discussed here, has two main facets: *re-structuring* a stovepipe system into components that could potentially be reused in other systems, and *re-engineering* the original system to permit replacement of certain components. One application is to upgrade portions of the stovepipe system to new technology, e.g., a new database management system or user interface toolkit. Another is to migrate to a new architecture, for instance, converting to the client/server paradigm, with some old components appropriately encapsulated to work together with some new ones. In this paper, we describe our experience using software process technology to support both aspects of componentization. The paper can be viewed as a status report on the current and future research directions we described in a previous publication in this conference series, which presented our ideas and the implementation status as of July 1994 [10].

We are particularly concerned in this paper with process support for *componentization* of process-centered environments (PCEs), which involved breaking up a particular existing PCE into components and reusing some of its components in a variety of environment architectures and frameworks. We started implementing the existing PCE in 1987, then known as Marvel and now as Oz, and it currently consists of about 280k lines of C, lex and yacc code. Approximately 60 graduate and undergraduate students participated in the effort, most of them for only one semester as part of an independent study project for academic credit, a handful for several years as research assistants. An "exploratory programming" style was used, and the resulting PCE certainly qualifies as a stovepipe system.

We developed a series of two enactable (executable) process models, each intended to support both aspects of componentization. The first, OzMarvel, was implemented in Spring 1993 as a Marvel environment instance and then employed for nearly two years in our initial development of Oz. EmeraldCity came on-line in April 1995 as an Oz environment instance and has been used for ten months as of this writing (February 1996). Prior to these component-oriented processes, we employed since January 1992 a Marvel process called CMarvel, unconcerned with components, to support our continuing development of Marvel following nearly the process we had previously employed on bare Unix. have also developed a variety of other Marvel and Oz processes, for applications outside software development and maintenance — ranging from document processing to healthcare workflow.

There were two main reasons for upgrading from OzMarvel to EmeraldCity. One was to bootstrap from Marvel to Oz as our platform to continue development of Oz. The Oz project is devoted in large part to componentization issues, while the predecessor Marvel project was not. Another important distinction between Marvel and Oz, for the purposes of this paper, is that a Marvel environment instance supports a single process that must be enacted by all users of that environment, whereas an Oz environment instance supports interoperability among multiple processes. A Marvel environment with an in-progress process can be converted to a single-process Oz environment, but as explained later EmeraldCity needed to exploit Oz's multi-process support.

EmeraldCity is also substantially different from OzMarvel in several other dimensions due to our early experience using OzMarvel to divide a relatively monolithic system into components and integrate experimental systems from those and other components. Thus the second reason was to incorporate what we had learned from our initial, relatively naive attempt at a component-oriented process and continue our re-engineering work with the significantly better process (i.e., from the viewpoint of componentization).

It may be confusing that we used a PCE to support componentization of that same PCE. There is nothing specific to PCEs in either of our two maintenance processes, so the approach should apply equally well to other medium-sized systems — but a PCE happened to be the system we were componentizing and from which our experience is drawn. That is, this was our real work, not an invented "case study".

First we provide some background on the Marvel and Oz process modeling and enactment systems. Then we describe the OzMarvel and EmeraldCity processes, with some discussion on how we used each of them in our componentization efforts. The paper concludes by summarizing lessons learned.

2 Marvel and Oz Background

Marvel [6, 13] and Oz [4, 2] employ client/server architectures. Clients provide the graphical user interface and invoke external tools. Servers context-switch among multiple clients, and include the process engine, object management, and transaction management for concurrency control and failure recovery (transaction management details are not addressed in this paper).

Marvel and Oz employ nearly the same rule-based process modeling language in which to define new processes or tailor reusable processes for an organization or project. A rule generally corresponds to an individual software development task, and specifies the task's name as it appears in a user menu; typed parameters and bindings of local variables to the results of queries on the project objectbase; a condition on the parameters and local variables that must be satisfied before initiating the activity — generally an external tool invocation — to be performed during the task; the tool envelope and arguments for that activity; and a set of effects, one of which asserts the actual results of completing the activity on the objects referred to in the parameters and variables. There is generally more than one possible effect if the tool has more than one possible result (the simplest example is a compiler than generates either object code or syntax error messages).

The process engine enforces that rule conditions are satisfied, and automates the process via forward and backward chaining. When a user requests to perform a task whose condition is not currently satisfied, the system automatically backward chains to attempt to execute other rules whose effects may satisfy the condition; if all possibilities become exhausted, the user is informed that it is not possible to enact the chosen task at this time. When a rule's activity completes, its asserted effect triggers automatic enactment of other rules whose conditions have now become satisfied. Both backward and forward chaining procedures operate recursively. Users usually control process performance by selecting a rule representing the an entry point into a task consisting of a one main rule and a small number of other auxiliary rules (reached via chaining) to propagate changes, but it is possible to define an entire process as a single goal-driven or event-driven chain — which is useful for simulation or training purposes. Built-in operations such as add an object, delete an object, etc., are modeled as rules for a uniform approach. Oz provides means for modeling and enacting synchronous and asynchronous "groupware" tools [19, 5], which were not supported by Marvel, but the details aren't relevant to this paper.

Marvel and Oz support nearly the same object-oriented data definition and query languages. A class specifies primitive attributes (integers, strings, timestamps, etc.), file attributes (pathnames to files in an intentionally opaque "hidden file system" that should not be accessed except through the PCE), composite attributes in an aggregation hierarchy, and reference attributes allowing arbitrary 1-to-N relations among objects, and one or more superclasses from which it inherits attributes (and rules treated as multi-methods [1]). Ad hoc and embedded (in rules) queries may combine navigational and associative clauses in a declarative style. Rules perform all data manipulation. Commercial off-the-shelf tools and other external application programs are interfaced to an environment instance through shell script envelopes, using augmented notation that hides from tool integrators the details of accessing the "hidden file system" and passing input and output parameters [9]. A return code from the envelope determines which of the several rule effects is asserted.

A Marvel environment consists of an arbitrary number of clients connected via an interprocess communication layer to a central server. Each server enacts one process, in which all its clients participate. Every client maintains its own objectbase "image" for browsing, which includes composite and reference attributes without primitive attributes and files, which are transferred only as needed — so the "image" is relatively small. In contrast, an Oz environment consists of one or more servers (termed "sites"), each with its own process model, data schema, objectbase and tools. Clients are always connected to their one "local" server (necessarily on the same local area network sharing a network file system), and may also open and close connections on user command to "remote" servers (which may, but need not, reside in other Internet domains with no shared file system). Servers communicate among themselves to establish *Treaties* — agreed-upon shared subprocesses automatically added on to each affected local process, and to coordinate *Summits* — enactment of Treaty-defined process segments that involve data and/or local clients from multiple sites. We stretch the International Alliance metaphor a bit, since Treaties among sites precede and specify Summits rather than vice versa.

3 First Try: OzMarvel

OzMarvel was our first component-oriented process. We used it to assist us in pulling subsystems out of Marvel to rewrite them into components, and at the same time implement Oz by direct extensions to Marvel — thus the name OzMarvel (our document processing environment was named DocMarvel, etc.). The main components envisioned were the process engine, the transaction manager, and the object management system. The long-term plan was to eventually reconstitute Oz from these components (the final phase, replacement of the process engine, is now in progress using EmeraldCity), after performing a set of experiments concerned with integrating some of our components into externally developed systems (notably Cap Gemini's ProcessWEAVER process-centered environment framework [8] and University of Wisconsin's Exodus database management system [7]) and replacing portions of Oz with externally developed components (notably the object management system from GIE Emeraude's PCTE industry-standard environment framework [18]); these experiments are discussed in [11, 16, 12, 15].

The OzMarvel data schema structures the objectbase into two main parts. One part consists of a set of teams, each consisting in turn of a group of private programmer workspaces. We used only two teams, representing current and past project members, respectively, but the schema allows for an arbitrary number. A workspace contains a set of C source and header files reserved by that user, locally generated object code and executables, and references to libraries in a shared repository needed to compile and build local executables. There are also means for testing with executables from other private workspaces (e.g., one user might be working on a new client while another works on a new server that must be tested together due to a change in the client/server protocol).

The other part of the objectbase consists of a set of projects, each representing a shared code repository. We had three, representing the baseline versions of Oz and its main Oz components, work progressing independently of Oz, and a frozen copy of all code delivered to a funding agency. The first two projects are collectively referred to as the "Master Area". Each project consists of a set of what the schema calls systems, a component pool, a module pool, and a pool of external libraries. Each system consists of a set of subsystems, each in turn corresponding to a distinct executable (a distributed system may involve multiple cooperating executable programs). For example, at the time we migrated the Oz code out of OzMarvel it had 19 subsystems: three variants of the server, four kinds of client, three translators for different Oz notations, the schema/process evolution tool, the daemon for automatically bringing up the server when a client starts up, and several utilities for managing an environment instance.

Libraries represent object code archives (i.e., Unix ".a" files) that may be linked into subsystems or components, together with their header files needed for compilation of importing code. For instance, OzMarvel had external libraries for `gdbm` (used as the backend of Oz's native object management system), for the PCTE object management system (which replaced `gdbm` in the two variant Oz servers), and for the `socks` secure TCP/IP sockets package (for authorized tunneling through corporate "firewalls"), along with `motif`, `xview`, `termcap`, etc. libraries imported by particular Oz clients.

Each subsystem referenced the several *context-free* components and external libraries (in the component and library pools, respectively) from which it was constructed, plus special-purpose modules for "glueing" those components and libraries together to construct the specific subsystem. The components in turn referenced the *context-free* modules (in the module pool) from which they were composed, and also contained local "glue" files for tailoring its modules to provide the functionality needed for that component. Each module (which could be and often was decomposed into a hierarchy of submodules) contained attributes representing its source files, object code archive, and public (to using modules) and private (for use only within the module) header files, as well as references to other header files needed for compilation. We emphasize context-free here, meaning that the components, modules, etc. were not supposed to make any assumptions about the systems and subsystems in which they were to be used and, at least in principle, were amenable to "plug-n-play".

Over its lifetime, OzMarvel was actively used by 14 people (not all at the same time). Although the basic philosophy and design remained the same, OzMarvel was modified several times to fix bugs in the process and to improve multi-user support; see [3] for a brief discussion of the schema and process evolution utility, called Evolver, used by MARVEL and Oz to upgrade the state of an in-progress process to match the semantic constraints of a new process model. The final process evolution left OzMarvel with 139 rules (only 26 task names appeared in the user menu, due in part to overloading — for instance, the `edit` command applies to many different kinds of objects — and in part to the marking of 75 rules in the process model as for internal propagation purposes only); 48 classes (13 of them virtual superclasses, such as `VERSIONABLE`, which would never be instantiated); and 37 tool envelopes.

Unfortunately, OzMarvel's multi-level structure proved much too complicated, evidenced by the relatively large proportion of propagation rules needed. For example, OzMarvel's rules to automate maintenance of each source file object's set of references to the objects representing each of the directly or transitively included header files were particularly intricate (and buggy). A header

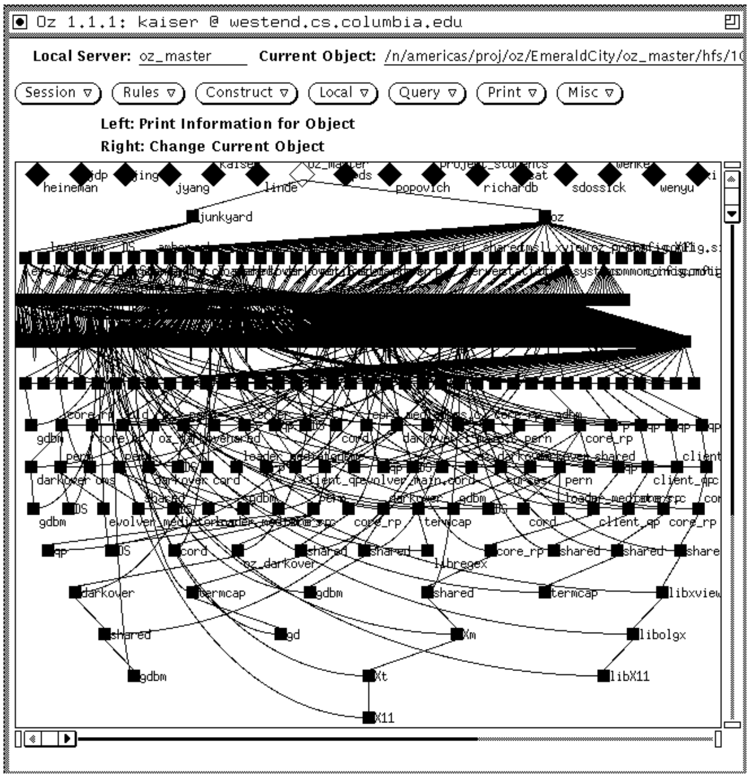

Figure 1: Hierarchical Master Area Display

file might include other header files with arbitrary recursion depth, and propagation rules were triggered whenever the source file or one of the header files was edited in a way that affected header file inclusion. Some of these references were different for each component/subsystem context in which the source file was used, since the same component might provide a different interface to different subsystems. This possibility was explained in our previous paper.

Further, multiple modules performed the same function with intentionally the same interface, i.e., there were at least two each of the major modules of the process engine, transaction manager and object management system in the module pool, corresponding to the original native modules in Oz vs. our new components. Thus the tools we had used for code cross-referencing in the previous CMarvel environment, standard Unix `etags` and our home-grown `revtags`, which assume a flat name space, did not operate properly in OzMarvel. Renaming solved this problem, e.g., *component-name_subroutine-name*, but made it difficult to plug-replace one component with another since code had to be edited (or preprocessed) for each subsystem context.

4 Second Try: EmeraldCity

When we wrote the previous paper, we had invented the name EmeraldCity, and intended that the new process would solve most of the problems with OzMarvel, but design was at a preliminary stage. At that time, we imagined that EmeraldCity would be constructed by evolving OzMarvel, but later we decided to construct the new process pretty much from scratch — although portions of OzMarvel's data model were retained.

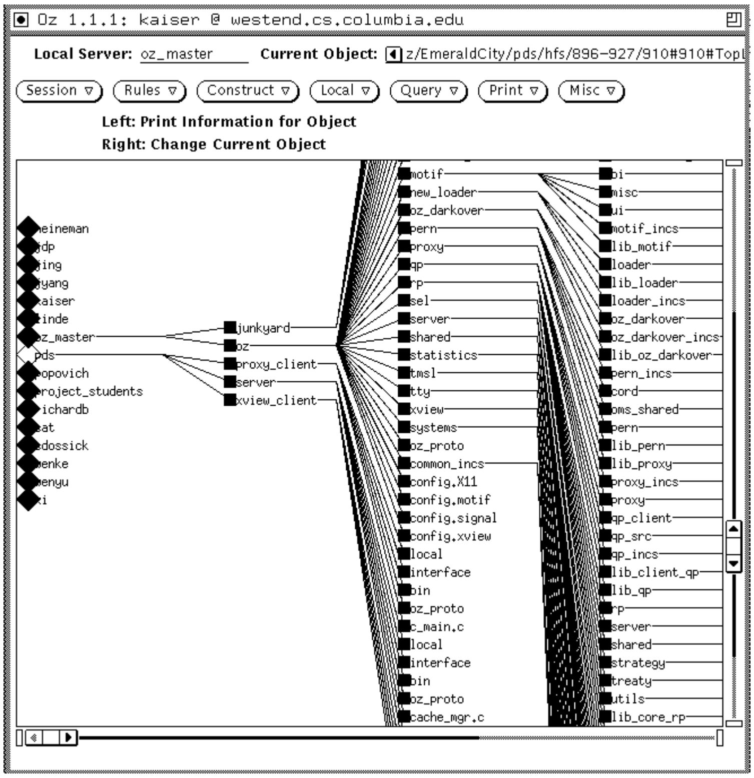

Figure 2: Horizontal Master Area and Workspace Display

EmeraldCity is really a set of several processes that work together, following Oz's International Alliance metaphor, rather than a single process like OzMarvel. EmeraldCity consists of two shared sites and an arbitrary number of workspace sites (16 at present, the number varies as students join and leave the project — or clone their workspace to perform relatively independent development in each one). Each EmeraldCity workspace has its own objectbase and process, whereas all OzMarvel workspaces necessarily are part of the same objectbase and share the same process. Thus moving to EmeraldCity gains advantages in performance (transfer of smaller objectbase images) and fault-tolerance (no central point of failure). EmeraldCity workspaces can be (and have been) shared by multiple users, but usually they are personal. One of the shared sites corresponds to the "Master Area" in OzMarvel, whereas the other "Assembly Area" is used only while a major re-engineering effort is in progress. Figure 1 shows the (not terribly readable) hierarchical view from the "Master Area" site (`oz_master`), showing only the local objectbase. Figure 2 shows a (somewhat more readable) horizontal view from that site, with an open connection to the `pds` site (Peter Skopp's workspace).

Over the summer 1995 we converted Oz from its native pointer-based object management system to using a OID-based object-oriented database component, Darkover [14]. An OID (or object identifier) is a unique identifier represented as an integer. The native transaction manager had already been replaced with a component [11], a much simpler effort performed using OzMarvel, and work on the new process engine component was still progressing independently.

Source and header file objects slated for re-engineering were checked out of the Master Area into a workspace (via a Summit) for changes, and then checked into either the Assembly Area (another Summit). Conversion efforts in relevant workspaces ranged from using home-grown tools that

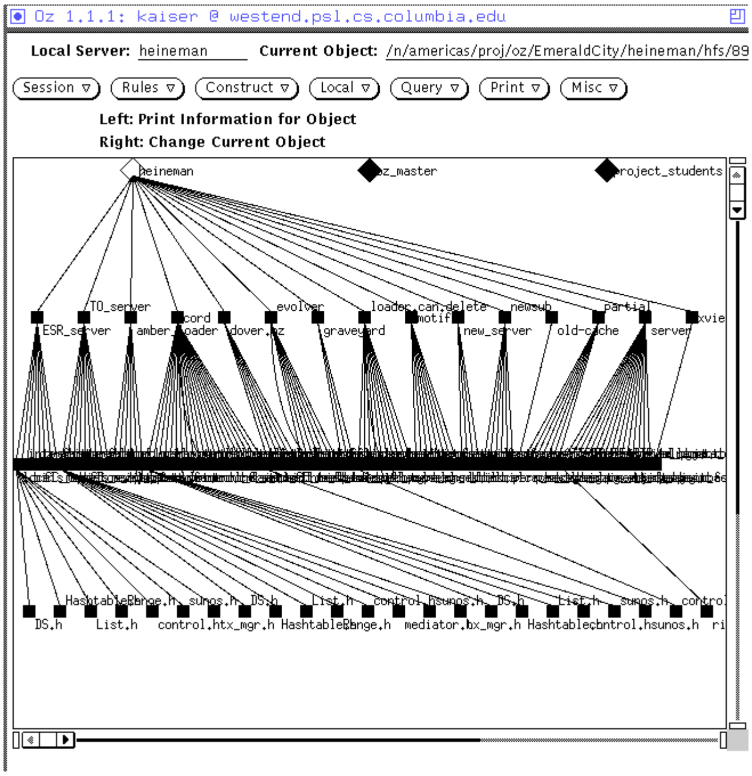

Figure 3: Workspace Display

semi-automated the lexical aspects of interface changes by matching code patterns that should be replaced with calls to Darkover's application programming interface; to recoding individual subroutines to traverse OID arrays rather than linked lists pointing to child objects; to module redesign, e.g., of Oz's cache manager for remote objects, which involved modifying Darkover to support transient objects — which probably would not have been possible had we been integrating with a foreign object management component.

The Assembly Area process allowed only completely converted code to be checked in. However, other code could still be checked into the Master Area, permitting unrelated development of portions of the system. This was very useful since not all of the developers were involved in the re-engineering effort, but had other pressing work to do that we wished to disrupt as little as possible. (The issue of concurrent re-engineering and development is a major concern of our previous paper.) Figure 3 shows the hierarchical view from the `heineman` site, which allows opening of only the "Master Area" and "Assembly Area" (called `proj_students` for obscure historical reasons).

Subsystem builds in each re-engineering workspace looked for non-local object code first in the Assembly Area and, only if not found there, in the Master Area; other workspaces were unaware of the Assembly Area. Treaties between Oz sites are set up on a pairwise basis that is neither symmetric nor transitive, so the connection graph need not be complete, although Summits can involve any number of sites that have agreed to the same Treaty. Our goal was to always be able to perform recompilation and build throughout the three months or so while the 150k affected lines (out of about 280k) were converted; this incremental approach would not have been possible without the binary compatibility of the old and new interfaces, due to C's allowance of type casting between integers and pointers. After the re-engineering effort was over, the entire code base was copied from the Assembly Area to the Master Area.

We have also been incrementally re-engineering our previously K&R C code base to ANSI-standard, the main focus of our previous paper. Although we settled on a mechanism somewhat more complicated than what was presented there, the effort has proved considerably simpler than the object management system replacement. After an initial flurry to convert the Master Area baseline, the work has proceeded more gradually: any C code (from OzMarvel or elsewhere) may be immigrated using Ozify (the Oz version of Marvelizer [17]) into an EmeraldCity workspace. There it is converted using a combination of the Gnu `protoize` tool, extensions to the envelopes of other tools, and manual changes to header files to preserve the conventions discussed below. The Master Area enforces that only ANSI-compliant code can be deposited (i.e., the ANSI C compiler using the strictest options generates no error or warning messages).

EmeraldCity restricts the contents of header files to avoid transitive dependencies, simplifies OzMarvel's notion of pools, and distinguishes *context-free* from *context-sensitive* representations of components. A project is composed only of a set of systems, a single prototype header file (`oz-proto.h`) included by all other header files in the project, a set of header files containing type definitions (but no prototypes) that may be used throughout the project, and a "program unit" pool. A prototype is essentially a forward declaration of a C function signature, as it must be used in source files whose object code will link with that function's code; prototypes are a required feature of ANSI C. `oz-proto.h` is automatically constructed by concatenating the contents of the `proto.h` header file associated with every library, which are in turn constructed automatically by tool envelopes as files are edited and compiled and libraries archived. Sections of `oz-proto.h` are guarded with preprocessor variables, so that only the relevant subset of the prototypes are used during compilation and there are no naming conflicts.

An EmeraldCity system consists of only a set of subsystems, as illustrated in Figure 4. A subsystem consists of a set of *context-specific* components, an executable, an archive for "glue" code between the components, and the source and object code for the "main" file (required by C convention for every executable). Each *context-sensitive* component contains source files for "glue" code, a reference to the single library representing the entire component, and a hierarchy of sub-components. These components are not reused in multiple subsystems — thus the designation *context-sensitive*.

In contrast, an EmeraldCity program unit is analogous to a *context-free* component in OzMarvel: it consists of a set of modules, a set of libraries, a set of local header files, and a set of references to header files in other program units. Each module contains its source files, and references the one archive holding its object code and the appropriate header files from its program unit; the modules assume the context of their program unit, but not of a subsystem component. Each workspace consists of a set of "local projects", which organize checked out files according to subsystem contexts and replicate relevant header files.

This new organization solved the naming difficulties that permeated OzMarvel. The files scanned by cross-referencing tools, `etags` and `revtags`, are always encapsulated in the appropriate context. We recently added a home-grown tool, called Hi-C, to generate HTML (HyperText Markup Language) to enable EmeraldCity users to view code and follow automatically generated hypertext links using World Wide Web browsers.

EmeraldCity has been actively used by 13 people to date. The Master Area site now consists of 78 rules (26 distinct names visible in the task menu, coincidentally 26 rules for internal propagation), 27 classes (6 of them virtual), and 32 tool envelopes. 21 of these rules are exported via Treaties to

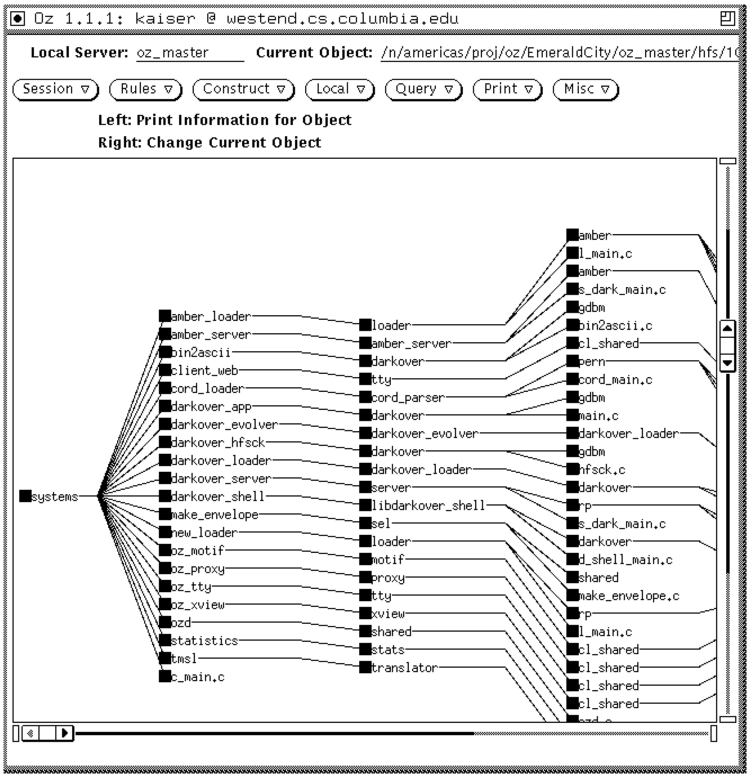

Figure 4: Zoom Into Systems Hierarchy in Master Area

workspaces for use in checkin/checkout, local build, etc. Workspaces are virtually identical to each other, although they need not be, with the main customization in the past being whether or not they formed Treaties (which have now been revoked) with the Assembly Area. A typical workspace has 68 rules (24 task names, 19 propagation), the identical 27 classes as the Master Area (this is <u>not</u> a requirement of Oz: different sites may have different schemas, with compatible subschemas needed only to match any Treaty subprocesses), and (coincidentally) 27 envelopes. The Assembly Area is the same as a Workspace, except for three special rules that were used in the Darkover conversion and two rules exported for the (now-revoked) three-way Treaties with a re-engineering workspace and the Master Area. One of the former three rules is shown in Figure 5 and one of the latter two in Figure 6; the other rules are similar.

5 Conclusions

Between launching of OzMarvel and completion of immigration into EmeraldCity (using Ozify), our code base nearly doubled from 155k to 280k lines; this does not include any external libraries or systems, e.g., for X Windows or used in our integration experiments. Since then, we have consolidated and replaced code, with little growth in the Master Area as of Oz 1.1.1. While neither Marvel nor Oz are production-quality in the commercial sense, we have been using the technology on a daily basis for over four years and have licensed the technology to over 40 institutions.

Our reconstruction of Oz from components and experiments integrating components with/from commercial systems necessitated our development and use of two generations of component-oriented

```
# rule signature
convert_CLASS [?c:COMPILABLE, ?cf:PROTOTYPE]:

  # bindings of local variables to results of objectbase queries
  (and (exists LOCAL_PROJECT ?lp suchthat no_chain (ancestor [?lp ?c]))
       # Use local version of  prototype file
       (forall PROTOTYPE     ?LPT suchthat no_chain (member [?lp.proto ?LPT]))
       # Local HFILEs
       (forall INC         ?li suchthat no_chain (member [?lp.inc ?li]))
       # Installed Interface (from set_subsystem[] rule)
       (forall INC         ?ii suchthat no_chain (member [?lp.interface ?ii])))
  :
  # condition
  # If the C file has not yet been compiled, this rule can fire.
  # The compilation changes the status of the CFILE to Compiled on success.
  (and no_forward  (?ii.recompile_mod = false)
       no_chain    (?cf.Name = "CLASS_PTR")
       no_forward  (?li.recompile_mod = false))

  # activity
   CONVERSION_TOOLS converter ?c.contents ?c.compile_log ?c.object_code
                          ?c.proto ?li.directory ?ii.directory ?LPT.contents
                  "-DMOVING_CLASS -Wall" ?lp.sys_includes
                          ?lp.compiler_directives ?cf.contents

  # success and failure effects
  (and          (?c.compile_status = Compiled)
       no_chain (?c.object_time_stamp = CurrentTime));
  (?c.compile_status = ErrorCompiled);
```

Figure 5: Darkover Conversion Process Task

```
# Build with master Main file (in SUBSYSTEM or SYSTEM.common_main)
# rule signature
build[?lp:LOCAL_PROJECT, ?p:PROJECT, ?mp:LOCAL_PROJECT]:
  # bindings of local variables to results of objectbase queries
  (and (exists SUBSYSTEM ?s  suchthat (and no_chain (ancestor [?p ?s])
                                           no_chain (?s.Name = ?lp.subsystem)))
       (exists BIN        ?lb    suchthat no_chain (member   [?lp.bin ?lb]))
       (forall COMPILABLE ?C    suchthat
                 (or no_chain (member   [?lp.files ?C])
                     no_chain (member   [?mp.files ?C])))
       # verify that there is no local Main file
       (forall CFILE      ?lm    suchthat
                 (or (and no_chain (linkto   [?lp.main  ?lm])
                          no_chain (ancestor [?lp ?lm]))
                     (and no_chain (linkto   [?mp.main  ?lm])
                          no_chain (ancestor [?mp ?lm]))))
       # get master main file
       (forall CFILE      ?main  suchthat
                 (and no_chain (linkto [?s.main ?main])
                      no_chain (ancestor [?s ?main]))))
  :
  # condition
  (and  no_chain   (?lm.Name = "")          # Not-Exists condition
        no_forward (?C.compile_status = Compiled)
        no_chain   (?main.compile_status = Compiled))

  # activity
  # Use main object code with the SUBSYSTEM.object
   COMBINE_TOOLS build_local ?lb.executable ?lp.build_log
                 ?s.libraries ?s.build_order
                 ?s.med_libraries
                 ?s.object_code ?C.object_code

  # success and failure effects
  (?lb.build_status = Built);
  (?lb.build_status = NotBuilt);
```

Figure 6: Treaty Process Steps for 3-site Builds

process models. These focus on the nitty-gritty but mandatory details of code understanding and configuration management, and ignore upstream aspects of the lifecycle (which are currently performed off-line). Although some of the problems encountered in OzMarvel were due to peculiarities of C, we'd expect to run into analogous difficulties using most programming languages — given that few were designed with "plug-n-play" componentry in mind. Oz's support for process interoperability (Treaty and Summit) proved an immense boon to component-oriented re-engineering of our process-centered environment framework and we expect would apply similarly to other medium-sized stovepipe systems.

Acknowledgements

The preliminary version of OzMarvel was developed by Andrew Tong and Steve Popovich. Issy Ben-Shaul contributed to the design of EmeraldCity. Wenke Lee participated in the Darkover re-engineering. Marvel, Oz, Darkover and the transaction management component (Pern) are available **without support**; send email to MarvelUS@cs.columbia.edu for licensing information.

A condensed, earlier version of this paper was submitted as a 3-page position paper to the *10th International Software Process Workshop.*

References

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

[2] Israel Ben-Shaul and Gail E. Kaiser. *A Paradigm for Decentralized Process Modeling.* Kluwer Academic Publishers, Boston, 1995.

[3] Israel Z. Ben-Shaul and Gail E. Kaiser. Process evolution in the Marvel environment. In Wilhelm Schäfer, editor, *8th International Software Process Workshop: State of the Practice in Process Technology*, pages 104–106, Wadern, Germany, March 1993. Position paper.

[4] Israel Z. Ben-Shaul and Gail E. Kaiser. A paradigm for decentralized process modeling and its realization in the oz environment. In *16th International Conference on Software Engineering*, pages 179–188, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[5] Israel Z. Ben-Shaul and Gail E. Kaiser. Process support for synchronous groupware activities. Technical Report CUCS-002-95, Columbia University Department of Computer Science, January 1995.

[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] Michael J. Carey, David J. Dewitt, Goetz Graefe, David M. Haight, Joek E. Richardson, Daniel T. Schuh, Eugene J. Shekita, and Scott L. Vandenburg. The exodus extensible DBMS project: An overview. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, chapter 7.3, pages 474–499. Morgan Kaufman, San Mateo CA, 1990.

[8] Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, February 1993. IEEE Computer Society Press.

[9] 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.

[10] George T. Heineman and Gail E. Kaiser. Incremental process support for code reengineering. In *International Conference on Software Maintenance*, pages 282–290, Victoria BC, Canada, September 1994. IEEE Computer Society Press.

[11] George T. Heineman and Gail E. Kaiser. An architecture for integrating concurrency control into environment frameworks. In *17th International Conference on Software Engineering*, pages 305–313, Seattle WA, April 1995. ACM Press.

[12] George T. Heineman and Gail E. Kaiser. The CORD approach to extensible concurrency control. Technical Report CUCS-024-95 (revised), Columbia University Department of Computer Science, February 1996. Submitted for publication.

[13] 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.

[14] Programming Systems Lab. Darkover 1.0 manual. Technical Report CUCS-023-95e, Columbia University, Department of Computer Science, March 1995.

[15] Wenke Lee and Gail E. Kaiser. Interfacing Oz with the PCTE OMS. Technical Report CUCS-012-95, Columbia University, Department of Computer Science, July 1995.

[16] Steven S. Popovich and Gail E. Kaiser. Integrating an existing environment with a rule-based process server. Technical Report CUCS-004-95, Columbia University Department of Computer Science, August 1995.

[17] 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.

[18] Ian Thomas. PCTE interfaces: Supporting tools in software-engineering environments. *IEEE Software*, 6(6):15–23, November 1989.

[19] Giuseppe Valetto and Gail E. Kaiser. Enveloping sophisticated tools into computer-aided software engineering environments. In *IEEE 7th International Workshop on Computer-Aided Software Engineering*, pages 40–48, Toronto Ontario, Canada, July 1995.