Incremental Process Support for Code Reengineering

(Experience Report)

George T. Heineman
Gail E. Kaiser

Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
(212)-939-7085
Fax: (212)-666-0140
heineman@cs.columbia.edu

CUCS-008-94

Abstract

Reengineering a large code base can be a monumental task, and the situation becomes even worse if the code is concomitantly being modified. For the past two years, we have been using the MARVEL process centered environment (PCE) for all of our software development and are currently using it to develop the OZ PCE (MARVEL's successor). Towards this effort, we are reengineering OZ's code base to isolate the process engine, transaction manager, and object management system as separate components that can be mixed and matched in arbitrary systems. In this paper, we show how a PCE can guide and assist teams of users in carrying out code reengineering while allowing them to continue their normal code development. The key features to this approach are its incremental nature and the ability of the PCE to automate most of the tasks necessary to maintain the consistency of the code base.

Key words: Process Centered Environments, Componentization

1 The Problem

Reengineering a large code base can be a monumental task. This paper shows how process centered environments (PCEs) can assist and guide a team of developers in reengineering tasks while allowing them to continue their development tasks. We differentiate between these two types of tasks since development tasks add extra functionality to the code base or fix defects, while reengineering tasks often focus on reorganizing the code base with no new enhancements. Budget schedules and other constraints often make it necessary for both tasks to proceed concurrently.

We have been using the Marvel PCE for over two years for all of our software development towards the production of Marvel, and its successor, Oz; during this time, the source code has nearly tripled in size from 73,000 lines of code to almost 200,000. Figure 1 shows the evolutionary history of Oz. The last single-user implementation[1], Marvel 2.6, can be roughly divided into three "modules" – a graphical user interface (GUI), a rule processor engine (RP), and an object management system (OMS). A more advanced multi-user system was eventually implemented, Marvel 3.x, consisting of a client and a server communicating through an inter-process communication layer (IPC). Since functionality was distributed between the client and server, each client needed to know some information about the RP and the OMS, hence the smaller "rp" and "oms" modules in the client. The design of the server, meanwhile, was further complicated by a new transaction module, TX, created to support multiple users; unfortunately, TX interacted far too closely with the existing modules. Additional executables, such as a process evolution utility, Evolver, and an objectbase conversion utility, bin2ascii, also needed some, but not all, of RP and OMS.

Our goal towards developing Oz 1.0 was to create a system composed of replaceable components. That is, Oz would allow the OMS to be replaced by an arbitrary system of equivalent functionality. Another interesting possibility is the replacement of the central rule processing engine. We realized last year that this componentization effort would be thwarted by the interconnections of these modules plus the existence of multiple module instances; therefore, we decided to reengineer the code base into components. We still had to perform, however, a large amount of new implementation, so we couldn't halt our development efforts.

Unfortunately, our software development process, illustrated in Figure 2, was unable to provide any special support for such a reengineering effort. Marvel 3.0 (released in December of 1991) was the last version of Marvel produced using the standard set of unbundled Unix tools. A Marvel environment instance, `marvel-3.x`, was created by executing the Marvel 3.0 PCE with a process specification, called C/Marvel. From that point on, all of our software development efforts were entirely managed by this environment instance. The first new Marvel version produced in this way, 3.0.1, was installed (replacing the Marvel 3.0 PCE) and a cycle was created, each

[1]There were earlier versions 1.0 and 2.x.

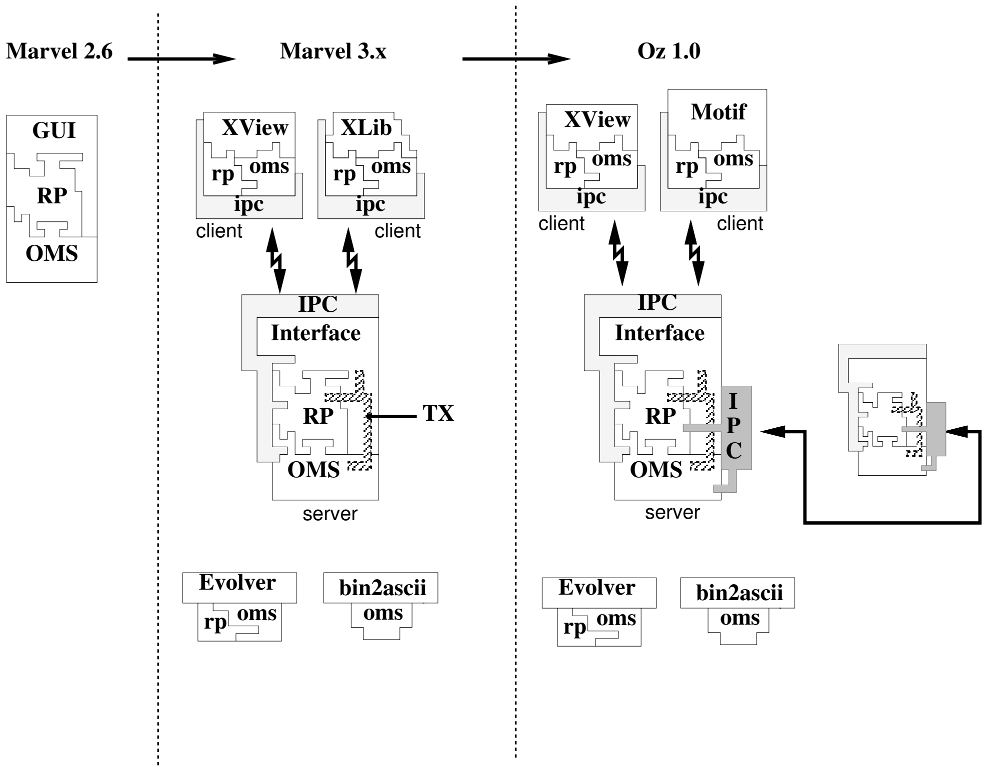

Figure 1: Evolution of Oz code

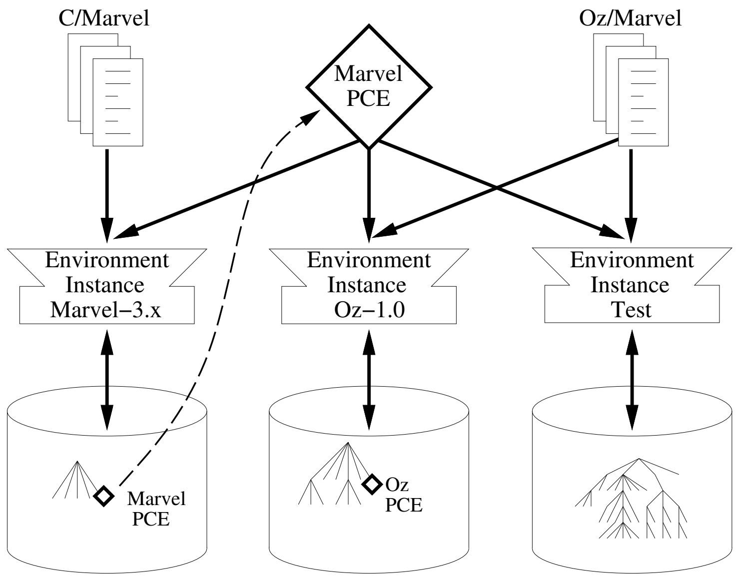

Figure 2: Marvel environments

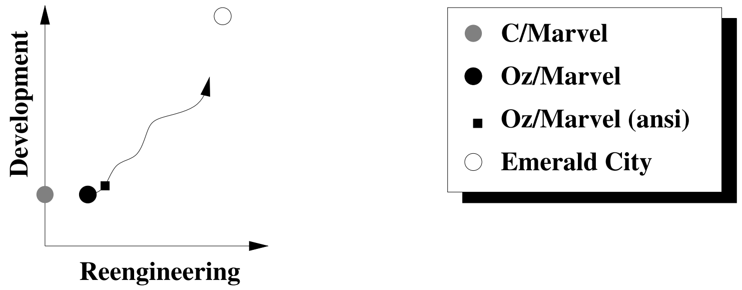

Figure 3: Reengineering vs. Development

new version being used to execute the C/Marvel process to help producing the next version. In this fashion, we produced four successive versions, culminating in Marvel 3.1.1. Since C/Marvel had no capability for the componentization we needed for Oz, we extended it into a new process called Oz/Marvel that was specially designed for our reengineering goal. We then were faced with the dilemma illustrated in Figure 3; either we could spend our time reorganizing the code into separate components in Oz/Marvel (the horizontal axis), or we could continue our development (the vertical axis) using C/Marvel, and mirror the changes slowly into Oz/Marvel. We decided to do both tasks simultaneously (the curving line), and this paper is the result of our experience.

The key to achieving both tasks is the controlled evolution of the process specification. Evolving C/Marvel to Oz/Marvel created a process specification with the beginnings of support for componentization. The ultimate goal, abstractly illustrated in Figure 3, is a process specification, that we call Emerald City, which will fully support componentized development. We first present a brief introduction to the Marvel 3.x system, followed by a short description of Oz/Marvel. We then describe in detail the modifications to Oz/Marvel that incrementally convert a code base from K&R C code to ANSI C[2]. This ANSI modification is a necessary first step towards our ultimate goal of componentizing a conventionally modular code base, and it provides insights into how PCEs can provide an *incremental* approach to reengineering.

2 Marvel PCE

The goal of the Marvel project [2, 6] is to develop a process centered environment that guides and assists teams of users collaborating on software development efforts. The behavior of the PCE is tailored by a process *administrator* who provides the schema, process model, and tool envelopes for a specific project. The administrator loads these specifications (written in Marvel's process modeling language, MSL, and

[2] K&R refers to C code conforming to the original C manual [10].

enveloping language, SEL) into the MARVEL PCE, creating a MARVEL *environment instance* that supports the data management, process management, and tool integration requirements of the project. The average MARVEL *user* follows the process in the environment instance, but is not involved in defining it. The object-oriented data schema is specified in terms of classes, each of which consists of a set of typed attributes. Attribute types include simple types, files, sets, and directed links. Set attributes contain instances of other classes, creating hierarchical composite objects. The set of all objects for an environment instance is referred to as an *objectbase*. Links are typed, allowing references between arbitrary objects as defined in the data schema. The C/MARVEL process mentioned earlier, for example, included 40 classes and 47 tool envelopes. Existing source code can be *immigrated* from the file system into a MARVEL objectbase using the Marvelizer utility [12]; when we "bootstrapped" the system to create the first `Marvel-3.x` environment instance, we used Marvelizer to migrate the MARVEL 3.0 source code into the `Marvel-3.x` objectbase.

The administrator defines the process (or workflow) by creating process steps corresponding to the individual software development tasks. Each step is encapsulated by a *rule* with a name and typed parameters. The body of a rule consists of a query to bind local variables (for example, the set of C code files that have not yet been inspected); a complex logical condition on the actual parameters and bound variables that must be satisfied prior to initiating the activity of the step; an optional activity in which a software development tool may be invoked; and a set of effects, each of which asserts one of the activity's possible results (if there is no activity, there is only one effect). The C/MARVEL process, for example, contained 184 rules.

MARVEL provides automated assistance by applying backward and forward chaining among the rules to automatically invoke process steps. Backward chaining is initiated to satisfy the condition of a user's request if it is not already satisfied. The backward chaining cycle completes when this condition is satisfied or all possibilities are exhausted. Forward chaining is carried out when the assertion of an effect of a rule makes the condition of another rule satisfied. The forward chaining cycle completes when all possibilities are exhausted. Both forward and backward chaining are recursive.

When the user requests a particular process step, MARVEL selects the "closest" matching rules (there may be more than one) and evaluates each of these rules in turn until it finds one whose condition is already satisfied or can be satisfied by backward chaining. If none of the conditions of the matching rules can be satisfied, the user is informed that it is not possible to undertake that process step at this time. If a satisfied rule is found, its activity, if any, is then executed. Afterwards, one of its effects is asserted, according to a status code returned by the activity, and MARVEL forward chains to any other rules that are implications of this effect.

Conventional file-oriented tools are integrated into a MARVEL process "as is" through envelopes written in SEL [5]. The MSL specification of the rule activity indicates the

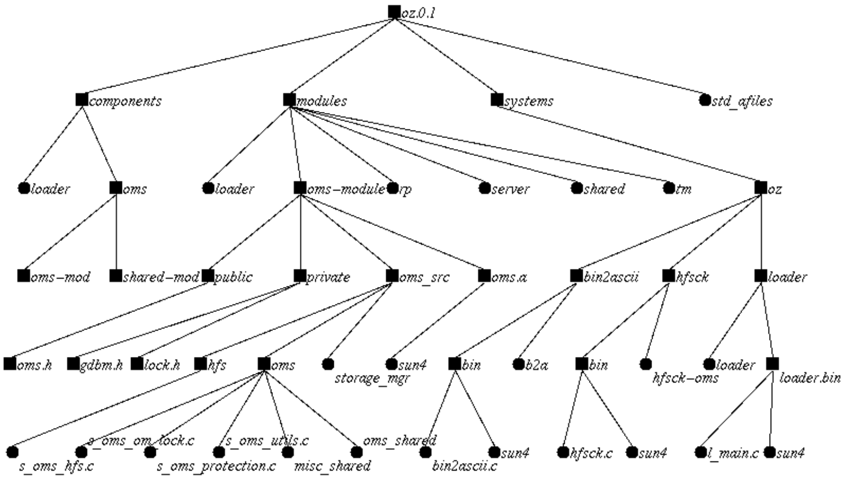

Figure 4: Objectbase Fragment

input and output arguments for an envelope; an implicit status code returned from the envelope selects the actual effect from among those given in the rule. The envelope header is written in SEL with its body, written in one of the conventional Unix shell languages: `sh`, `ksh`, or `csh`.

3 Componentization

The first step towards componentization is a suitable organization for the code base. The *stable area* specified by Oz/Marvel has three main subdivisions: the module, subsystem, and component pools. Each MODULE[3] object manages a single library archive file (for example, `oms.a`) that contains the object code for all source files that are descendents of the MODULE. MODULE objects also contain private and public HFILEs (representing `.h` header files); a MODULE's interface is defined by its public HFILEs. COMPONENT objects combine MODULE objects to provide a set of services. COMPONENTs do not produce executables, however; SUBSYSTEM objects integrate COMPONENTs and appropriate "glue" code to provide a single executable. A SYSTEM is a set of SUBSYSTEMs that work together.

The `oz-1.0` environment instance (from Figure 2) has been instantiated from the Oz/Marvel process specification. Figure 4 shows a subset of `oz-1.0`'s objectbase; each object drawn with a circle (instead of a square) has descendent children hidden for clarity. The subdivisions shown contain:

- Two COMPONENTs: `loader, oms`
- Six MODULEs: `loader, oms-module, rp, server, shared, tm`

[3]Oz/Marvel class specifications are in UPPER CASE.

- Three SUBSYSTEMs: `bin2ascii, hfsck, loader`

The functionality of the COMPONENTs is separated from the composition of the MODULEs through the use of link attributes (not shown in the figure); this allows MODULEs to be reused between COMPONENTs. The `oms` COMPONENT combines two MODULE objects with links from each of its children to the corresponding MODULE (that is, `oms-mod` links to the appropriate MODULE named `oms-module`). `oms-module` has one public HFILE, `oms.h`, and one private HFILE, `gdbm.h`. This encapsulation of private information is crucial in complex systems; here, the actual implementation of the object management system (that uses the GNU database manager for its byte server) is hidden from the COMPONENTs that use `oms-module`. Each object code that is a descendent of `oms-module` is archived in the `oms.a` archive file. Finally, the `oms` COMPONENT is used by two SUBSYSTEMs, `bin2ascii` (a utility that translates Oz binary objectbases into an ASCII readable format) and `hfsck` (a utility for verifying the internal file directory structure for the file attributes of an objectbase). They each have a child (`b2a` and `hfsck-oms`, respectively) with links to the `oms` COMPONENT, showing how SUBSYSTEM objects use the services provided by the COMPONENTs to produce their own executables. `hfsk.c`, for example, is the appropriate glue code for the `hfsck` SUBSYSTEM.

`oz-1.0` currently contains nearly 1800 objects as it manages the code base of over 300 source code files. The full Oz SYSTEM is composed of thirteen SUBSYSTEMs, including three Oz client interfaces (Motif, XView, and TTY), the Oz kernel, and supporting utility programs. The code base is protected from arbitrary modifications since all code changes occur in local workspaces rather than the stable area. When modified code is reintegrated into the stable area, Oz/Marvel automatically marks other objects that are affected, so that the libraries and executables of the code base can be later rebuilt with a minimum of effort.

Oz/Marvel also has additional features that make it more powerful than any fixed mechanism, such as `make`, that relies upon the timestamp of a file to determine the steps it needs to carry out. Each object has attributes, defined in the schema, that maintain information used by the process as it performs its tasks. Oz/Marvel reacts differently, for example, when a programmer modifies an HFILE to change the `#include` dependencies than when a simple source code change is made to the file; this is made possible by a SEL envelope, that determines the nature of the change once it has been made.

We have been using the Oz/Marvel process daily for over a year, and have noticed some interesting points as we have been modifying and improving it. The first is that Oz/Marvel is clearly a transitional process. Referring back to Figure 3, the Oz/Marvel process specification is being continuously evolved to reach the ultimate goal (in the upper right-hand corner) of becoming the Emerald City process specification. In Section 5, we discuss how managed evolution of the process works.

The second point of interest is the many ways in which Oz/Marvel supports our componentization effort. Individual COMPONENTs can be built and tested in a programmer's local WORKSPACE (another class of object in the same objectbase as the COMPONENT), allowing COMPONENTs to be developed separately before integration. Programmers can execute certain rules to produce a detailed list of functional dependencies, if any, that exist between COMPONENTs. This allows potential encapsulation problems to be detected early since "hidden" dependencies can be made visible. The process uses envelopes to maintain `TAGS` files, a common form of cross-referencing created by the `etags` program. We have extended the standard `TAGS` file to allow the process to determine the *use* (not just the definition) of a particular tag. The process can then use this information intelligently; for example, whenever a change occurs to a public function declaration in an HFILE, only the CFILE objects that actually use that particular function need to be updated (a simple form of smart recompilation [14]).

One question to be answered is whether the overhead required by a PCE such as ours can be validated. There are several problems, however, when we attempt to address this question. For example, it is not easy to quantify the extra work performed by the PCE when executing the Oz/Marvel process, since we have no standard benchmark against which to measure it. It is not clear, for example, how one could compare a PCE with working directly on Unix with no process support. Our own experience with Oz/Marvel has shown that it is invaluable to have a process engine capable of maintaining consistency when a number of programmers are working together. The Oz/Marvel process has constraints that make it impossible for a "novice" programmer to corrupt code in the stable area. For example, code cannot be deposited into the stable area unless it has compiled without errors and passed a code inspection review.

We now describe in detail an example of adding a process constraint, namely, that all code be made ANSI-compliant. This example provides a better understanding of how the Marvel PCE works and shows the incremental nature of our approach to reengineering.

4 Adding a process constraint

In Oz/Marvel, the code base is partitioned into one stable area and multiple personal workspaces, one for each user. Programmers reserve files into local workspaces, and check the files back into the stable area once all modifications are complete. This is just one example of a Marvel process; in particular, the check-out mechanism described is not inherent to all Marvel processes. We now describe the parts of Oz/Marvel specific to the ANSI conversion.

Objects representing C source code belong to the CFILE class, while those representing header files (`.h` files) belong to the HFILE class; the Oz/Marvel definitions for these

```
HFILE :: superclass REFERENCED, FILE;
   ansi        : boolean = false;            ## new for ANSI
   has_ansi    : boolean = false;            ## new for ANSI
   ansi_version : text = ".ansi.h";          ## new for ANSI
   contents    : text = ".h";
   action      : (None, ChangedFile, ChangedDependency) = None;
end

CFILE :: superclass INSPECTED, REFERENCED, FILE;
   contents   : text = ".c";
   machines   : set_of MACHINE_SRC;
   converting : boolean = false;             ## new for ANSI
   ansi       : boolean = false;             ## new for ANSI
end
```

Figure 5: Oz/Marvel definitions for HFILE and CFILE

classes are shown in Figure 5. The header code for an HFILE object is stored in its *contents* attribute, a `text` attribute representing a file on the file system. The `boolean` attribute *ansi* indicates whether the HFILE object has been fully converted to ANSI, while *has_ansi* is `true` when the HFILE has been partially converted. This partial ANSI version is found in an HFILE's `text` attribute, *ansi_version*, for use in compilation by ANSI CFILEs. A partially converted HFILE has some of its functional prototypes converted to the ANSI standard; ANSI-C compilers are backward compatible in that they accept non-ANSI functional prototypes. The original non-ANSI version is kept in *contents* for use by non-ANSI CFILE objects. CFILE objects are ANSI-compliant if their *ansi* attribute is `true`. The `boolean` attribute *converting* is used during chaining when a CFILE object is deposited in the stable area.

The programmers develop their code using the process fragment illustrated in Figure 6. Each individual node represents a rule with its typed parameters. Single-width arrows represent forward chains between rules while thick lines show backward chains. The dashed lines show sequences that are logically imposed on the user, but not automated; for example, only the user knows when a CFILE is ready to be deposited. The rule chains shown in this process provide the following support:

- If a programmer changes the list of `#include` HFILEs for a CFILE object, a forward chain to local_ref[CFILE] is initiated to recompute the transitive closure of HFILE dependencies.
- If a programmer makes a change to a CFILE object, the system checks it for errors using a static program analyzer before invoking the compile process step.
- If a programmer modifies an HFILE object, all CFILE objects in the same local WORKSPACE are marked as `NotAnalyzed` (preparing them for the analyze[CFILE] process step). If this modification changed `#include` HFILE dependencies, the CFILE objects are marked as `NotReferenced` (preparing them for local_ref[CFILE]).

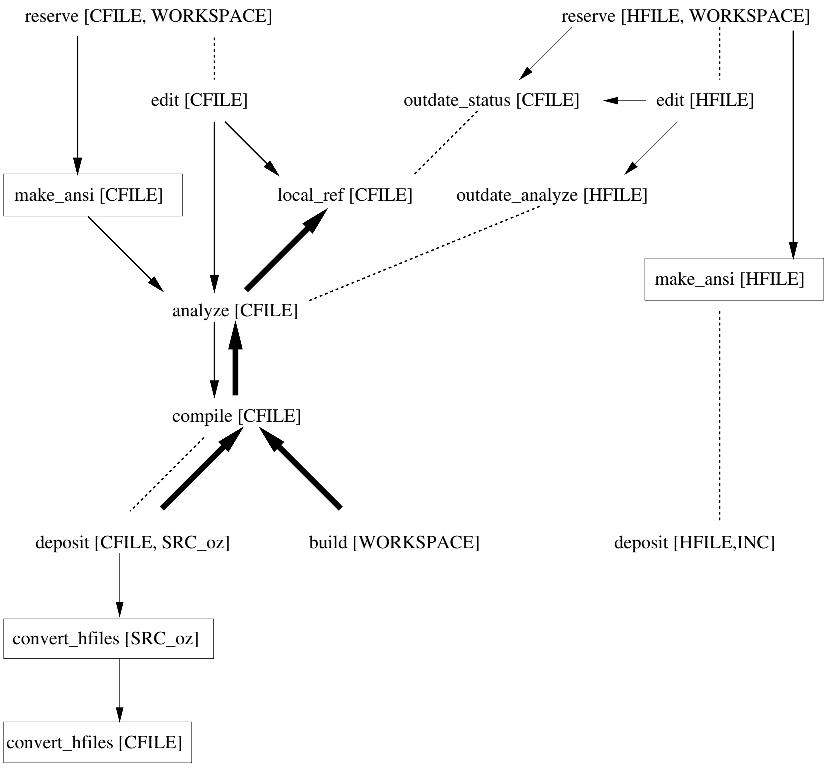

Figure 6: Oz/Marvel Process Fragment

The new rules (shown in boxes) add the following support:

+ When a programmer reserves a CFILE, the process automatically converts it to ANSI-C by forward chaining to make_ansi[CFILE]. The activity of this rule invokes the `convert_cfile` envelope, shown in Figure 8. This envelope uses the `protoize` tool, provided with the GNU C compiler, to convert existing function declarations and definitions in CFILEs and HFILEs into ANSI-C format.
+ When a programmer reserves an HFILE, the process marks it as ANSI only if all the CFILE objects that `#include` it have been converted to ANSI.
+ When a programmer deposits a CFILE, the process automatically updates the functional prototypes found in HFILE objects in the stable area and marks these HFILE objects as partially converted (the *has_ansi* attribute from Figure 5).

The process change to Oz/Marvel, introducing the last three features, was limited to four new rules, five new attributes to existing classes, and two new envelopes. Quantitatively, this change required 158 lines of shell envelopes and 97 lines of MSL, a change of less than 4% to the original Oz/Marvel process. This modified process will ultimately convert an entire code base if each individual CFILE is eventually reserved into a local WORKSPACE[4]. A balance must be reached that maximizes the

[4]This could easily be extended to allow administrators to convert CFILEs in the stable area.

```
make_ansi[?c:CFILE]:
  # Chain into member[] predicate when a new CFILE appears under WORKSPACE
  (and (exists WORKSPACE       ?w suchthat           (member [?w.files    ?c]))
       (exists HFILE           ?h suchthat  no_chain (linkto [?c.hfiles   ?h]))
       (exists MACHINE_SRC_oz  ?m suchthat  no_chain (member [?c.machines ?m]))):

  no_chain (?c.ansi = false)

  { ANSI_tools convert_cfile ?c.contents ?c.history ?h.contents }

  (and (?c.ansi = true)                         # Successful conversion
       (?c.converting = true)
       (?m.analyze_status = NotAnalyzed)
       (?m.compile_status = NotCompiled));
  no_assertion;                                 # Unsuccessful
```

Figure 7: Oz/MARVEL make_ansi[CFILE] rule

```
ENVELOPE convert_cfile;
SHELL sh;
INPUT
          text : cfile;
          text : log;
   set_of text : hfiles;
OUTPUT none;
BEGIN
  TEMP=/tmp/convert_cfile$$           ## Create temp place in which to work
  mkdir $TEMP ; cd $TEMP

  for i in $hfiles                    ## link to HFILEs here
  do
    ln -s $i
  done

  RET_VAL=0                           ## Assume will work correctly
  protoize -g -k $cfile > $log        ## Protoize the CFILE.  [-g] includes
  if [ $? -ne 0 ]                     ## prototype information inside the CFILE
  then                                ## source code.  [-k] keeps the *.X file,
    RET_VAL=1                         ## that we use later
  fi
  cd /tmp ; rm -fr $TEMP              ## Clean up
RETURN "$RET_VAL";
END
```

Figure 8: Oz/MARVEL convert_cfile envelope

effectiveness of automatic tools and the availability of human resources. For example, the `protoize` tool does not automatically convert C functions that contain a variable number of arguments; these must be manually converted by the programmer to the ANSI `stdarg` convention.

The process change we have just described in detail is only a small step towards our effort of componentizing a code base. The declarative nature of MARVEL's rule-based process modeling language makes it possible for an administrator to add on new process fragments and remove outdated parts using the Evolver utility. For example, once the entire code base becomes ANSI-compliant, the four new rules will no longer be applicable, and could be removed entirely or simply disabled. In the following section, we outline our approach towards creating EMERALD CITY, a process specification that fully supports componentized development.

5 Emerald City

As we have seen, a PCE can take a process specification and create an environment instance supporting that process. Our reengineering goal can be met by producing EMERALD CITY, a process specification that supports both development and componentization. Components are different from programming modules in several ways. Modules are usually small, created to encapsulate data or procedural abstractions; their interface contains a handful of operations. A typical example is a queuing package that hides the actual details of how it implements queues. Most likely, modules cannot stand by themselves and must be linked with other modules. Modules can be replaced, in theory, but, since their interface is small, this is normally not done. Components are large, created to provide a set of services. They group modules together, creating a stand-alone entity which can communicate with other components or other system layers. A typical example is an object management system. Components do not link together as closely as modules do, and often require some "glue" between them to match interface specifications. Components are large enough to consider building a system with replacable components; for example, completely replacing the underlying object management system.

We are creating EMERALD CITY by adding constraints to the OZ/MARVEL process specification; the mechanism we use is the Evolver utility. Ealier, we showed an example of the first such constraint, converting K&R C code to ANSI. This was added to strongly type the code in an attempt to find functional mismatches (i.e., functions called with the incorrect number or type of arguments). OZ/MARVEL also has rules which produce a list of functional dependencies between COMPONENT objects, itemizing the breaches in encapsulation. The developers can then reorganize the code or extend the appropriate module interfaces as necessary.

Our most pressing concern is allowing COMPONENTs to have different interfaces depending upon the particular SUBSYSTEM which is using it. Figure 3, for example, shows how the `oms` COMPONENT is used differently in four different SUBSYSTEMs. This ability is a prerequisite for entirely replacing a COMPONENT from a set of SUBSYSTEMs (i.e., a SYSTEM). We are currently extending Oz/Marvel to provide this ability.

6 Related work

Many existing PCEs use some form of rules to define processes because declarative rules are believed by many researchers to be the most natural way to express (at least the local) constraints on software development tasks. Major exceptions include Arcadia [8], which uses an imperative notation called APPL/A [13] based on Ada, HFSP [9], which uses an extension of attribute grammars, and Melmac [3], Slang [1], and Process Weaver [4], which use a form of Petri nets. Important examples of rule-based PCEs include Grapple [7] and Merlin [11].

We would like to compare our experience against the following two concerns: How (if at all) have the implementors of PCEs been able to support reengineering of their own code? What processes (if any) have been constructed to assist and guide users in their reengineering efforts? This comparison must remain open since we have not found any other similar experiences in the literature. We anticipate, however, that the experience from this paper should be most applicable to rule-based and petri-net based PCEs.

7 Conclusion

For our approach to be successful, it must have two features. First, the PCE must be able to automate most of the work. The programmers should not have to perform extra steps manually otherwise they will choose not to use the process. Second, the approach must be incremental; an off-line approach will not work since the reengineering and development tasks occur concurrently. In this paper, we have shown how the Marvel PCE can be used to execute a process specification (Oz/Marvel) that assists and guides the developers as they concurrently perform their reengineering and development tasks.

There are situations, however, under which our approach is not valid. Some reengineering efforts are not automatable, such as a complete redesign using C++ instead of C. In these cases, an appropriate supporting process would be more interactive, assigning more responsibility to the programmers. Alternatively, code that has already been well designed and implemented might not even need support for reengineering. Experience suggests, however, that most C code is organized poorly. This is a direct result of the simplicity of the C language (sometimes referred to as a high-level assembly language). The widespread use of Unix and C bypass all arguments for whether one should use C as a programming language. Since millions of lines of C code already exist, there is most definitely a need for supporting simultaneous reengineering and development of the same code.

We are currently using Oz/Marvel to componentize the Oz system. The primary work is aimed at isolating the three main components of the Oz code base, namely the rule processor, transaction manager, and object management system. The two goals of this research are to create an architecture with replaceable components and to reuse these components in other systems.

Acknowledgements

We would like to thank Andrew Tong for his prelimary work on Oz/Marvel and Israel Ben-Shaul for his insightful discussions on the transition from Oz/Marvel to Emerald City.

References

[1] Sergio Bandinelli and Alfonso Fuggetta. Computational reflection in software process modeling: the SLANG approach. In *15th International Conference on Software Engineering*, pages 144–154. IEEE Computer Society Press, May 1993.

[2] 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.

[3] Wolfgang Deiters and Volker Gruhn. Managing software processes in the environment MELMAC. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 193–205, Irvine CA, December 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[4] 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.

[5] 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.

[6] 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.

[7] Karen E. Huff and Victor R. Lesser. A plan-based intelligent assistant that supports the software development process. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 97–106, Boston MA, November 1988. ACM Press. Special issue of *Software Engineering Notes*, 13(5).

[8] R. Kadia. Issues encountered in building a flexible software development environment. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 169–180, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[9] Takuya Katayama. A hierarchical and functional software process description and its enaction. In *11th International Conference on Software Engineering*, pages 343–352, Pittsburgh PA, May 1989. IEEE Computer Society Press.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs NJ, 1978.

[11] Wilhelm Schäfer, Burkhard Peuschel, and Stefan Wolf. A knowledge-based software development environment supporting cooperative work. *International Journal on Software Engineering & Knowledge Engineering*, 2(1):79–106, March 1992.

[12] 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.

[13] S.M. Sutton, Jr. APPL/A*: A Prototype Language for Software-Process Programming*. PhD thesis, University of Colorado, August 1990.

[14] Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.