PAGE CONNECTION REPRESENTATION: AN
OBJECT-ORIENTED AND DYNAMIC
LANGUAGE FOR COMPLEX WEB
APPLICATIONS

A Thesis Presented to
The Faculty of the
Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University
In Partial Fulfillment
of the Requirement for the Degree
Master of Science

by
Yin Zhou
August, 2001
To My Dear Parents

Shenquan Zhou and Jiamei Hu
ACKNOWLEDGEMENTS

I would like to pay special thanks to my team members in PowerYP project: Xiaodong Zhang, Hong Chen, Lu Tong and Dazhang Gu. Without the unforgettable stimulating discussions, it would be impossible for the birth of PCR language and my current thesis. Another team member, Yanting Wang from University of Rochester, also should be thanked for his input to this project.

Finally, I want to thank my computer science advisor Dr. Liming Cai for his support during this project. Professor Cai gave me the freedom to follow my own research path and was always willing to discuss and speculate on the direction that my research was taking.
Contents

List of Figures ................................................... viii
List of Tables .................................................... x

1 Introduction ................................................. 1

2 General Framework: a Top-down Approach .......... 9
  2.1 Web serving mechanisms ................................. 9
    2.1.1 Static Page Serving ................................. 11
    2.1.2 Dynamic Page Serving ............................... 11
    2.1.3 Dynamics of HTTP Transaction for PCR generated application server 13
  2.2 Page Connection Representation of a Web Application 15
  2.3 Architecture of the Page Application Object ....... 17
    2.3.1 View ................................................. 20
    2.3.2 BLU ................................................ 20
    2.3.3 High Level Database Access Interface ............ 23
  2.4 Decompose the Page into Irreducible Components .... 23
3 Description of the PCR Language

3.1 Getting Started .................................................. 27
3.2 Structure of PCR Program ......................................... 31
3.3 Basic Types, Dynamic ADT, Static ADT and Reference Type ........ 34
3.4 LIST ................................................................. 38
3.5 Variables and Arithmetic Expressions .......................... 39
3.6 Statement ........................................................... 41
3.7 Flow Control ........................................................ 42
  3.7.1 Logical Expression/Condition ................................. 42
  3.7.2 IF Statement .................................................. 43
  3.7.3 Loops — FOR ................................................ 45
  3.7.4 Loops — WHILE ............................................. 46
  3.7.5 Loops — DO-WHILE ....................................... 46
3.8 Method ............................................................... 47
3.9 Database Specification File ....................................... 48
3.10 Database Connectivity ........................................... 49
3.11 Multiple Pages and Page Connections ......................... 51
3.12 Multiple Views ................................................... 55

4 Application of PCR to E-Magazine Management System(EMMS) 57

4.1 Database Image of EMMS ......................................... 57
4.2 Page Connection Representation of EMMS ....................... 60
4.3 Illustrated PCR Code Skeleton for EMMS ........................................ 62

5 Comparison of PCR with Related Web Technologies 69

5.1 Active Server Page and PCR .................................................. 69

5.2 XML and PCR ................................................................. 75
  5.2.1 XML as a Client Side Technology ............................... 76
  5.2.2 Fitting XML in PCR Scenario through Messaging ............... 81

6 Future Extension 84

6.1 Hyper Component Library .................................................. 84

6.2 Visual Development Environment On Web ............................ 87

7 Conclusion 89

A PCR compiler implementation 90

A.1 PCR Grammar ............................................................... 90
  A.1.1 PCR View Grammar ................................................... 90
  A.1.2 PCR Service Logic Grammar ....................................... 91

A.2 Symbol Table ................................................................. 105

A.3 Intermediate Code Generation ............................................ 106

Bibliography 108
# List of Figures

1.1 The Lifecycle of a Web Application ............................................. 7  
2.1 3-tier architecture of web applications ........................................... 10  
2.2 Static Page Serving ................................................................. 11  
2.3 Dynamic Page Serving ............................................................... 13  
2.4 Dynamics of HTTP Transaction for PCR generated Application Server ... 14  
2.5 Execution Sequence of Page Objects ............................................... 16  
2.6 The PCR’s Pointview of Web Application: a group of cooperating page objects ................................................................. 18  
2.7 The PCR’s Pointview of Single Page Object ....................................... 19  
2.8 The Diagram of View ................................................................. 21  
2.9 The Business Logic Unit ............................................................... 22  
2.10 Decompose a sample product catalogue page into smaller components ... 24  
3.1 The Correspondence between PCR source modules and target modules ... 33  
3.2 The illustration of state-machine behind switch program ..................... 55
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Database Image of EMMS</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Page Connection Presentation of EMMS</td>
<td>61</td>
</tr>
<tr>
<td>5.1</td>
<td>Using ASP and COM</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>The Fitting of XML in PCR scenario</td>
<td>82</td>
</tr>
<tr>
<td>5.3</td>
<td>Importing XML Message to Page Object</td>
<td>83</td>
</tr>
<tr>
<td>6.1</td>
<td>Hyper Code</td>
<td>85</td>
</tr>
<tr>
<td>6.2</td>
<td>Component Library of PCR</td>
<td>86</td>
</tr>
<tr>
<td>6.3</td>
<td>Visual Development Environment on Web</td>
<td>87</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Comparison of HTML and XML .............................................. 80
Chapter 1

Introduction

With the proliferation of dot coms in the end of last century and the revolutionary rise of the tidal wave of e-commerce\[1, 2\], the need to create complicated web applications and to speed up electronic publishing of all kinds of data is being increased dramatically. Among these web applications, most of them deal with dynamic web pages that could serve as interfaces for conventional databases\[3\]. Developing dynamic web pages does not seem so intimidating at its first look for a proficient programmer. But as the site goes more and more complex, the relation between dynamic web pages becomes more and more intriguing. Experiences accumulated from those maze-building processes start to identify the main cost of all e-projects today to be the human factor\[4, 5\]. Intense software development must be accomplished by sophisticated programmers in order to deploy and launch a large scale database backed dynamic website. But in an industry where even the speed of light still looks too slow, the law of relativity makes the demand for the development speed reach its extreme.
Replying to this issue, lots of development tools have been published since 1995 to deal with the dynamic web page development problem. Both client and server side solutions are provided. On the client side, Javascript, DHTML, CSS and VBScript provide the user with more freedom to interact with the page content. Java Applet and ActiveX, in the meanwhile, infuses the programmable components on client pages. In addition, Java wants to realize the philosophy of ubiquitous computing by enforcing the concept of “program once, use it everywhere” through the porting of JVM[6]. But for the intense competition in the client market, lots of technologies are incompatible from platform to platform. The most prominent phenomenon is the incompatibilities existing between Microsoft products and Netscape or other products originated from the Unix world. Till today, the most acceptable and robust client technologies are still pure HTML, some Javascripts and the newly adopted CSS. In the famous conflict between Sun and Microsoft, Java, though a great and delicate network language as it has been proven, has not been supported so well on all platforms, especially on Internet Explorer. Even for Javascript, Microsoft also released a similar competing product — JScript[7], in order to enforce its own “One Microsoft Way”. In fact, all these client technologies can be classified into two categories: one includes scripts, another includes network programs like Java. But because of the platform dependence of these technologies, a small porting problem could always be a big headache even for a decent programmer.

With the client war in place, since 1996 more and more web sites have resorted to server side technologies, which promise to provide a more intimate connection with the database and circumvent the portability problem by finishing all the sophisticated dynamic
computings on the server side before sending back a compatible hyper text with minimum client technology to afford a reasonably good web interface. This tendency up with the prevailing of the NC and NetPC, leads to a much cherishable thin-client concept. This is actually a revenge come-back of old Unix utilities like text-mode telnet and graphic-mode X window system on the web. After a long suppression under the popularity of PC, the web provide people a new viewpoint to look at and a new stage to exploit the old-fashioned thin-client concept evolving from its old terminal concept residing in old Unix techies' head.

The major movement afterwards on server side technologies could be classified into web servers, extended modules (cgi program, web scripts, server APIs) and the new server module development languages and environments [8, 9, 10, 12, 16]. For most e-commerce site developers or any other web application developers, the major concern is not the web server, but is the add-on features and services of their web sites, which are usually provided through extended modules. But all the technologies in this area have some advantages and disadvantages.

For the cgi program, it is easy to understand, but has the unnecessary process launching overhead for each user request. For web scripts like ASP[9], PHP[10] and server side Javascript[12], they eliminate the process launching overhead and achieve great scalability, but an interpreter module must be installed on the web server and the on-line parsing and interpretation of the script slows down the serving pace. Furthermore, the interpretive nature of the script also inhibits the optimization of the service code[13]. Another alternative dynamic web development tool, Cold Fusion, also faces the similar situation as the web scripts [14].
Besides these trade-offs, all these server-side technologies are facing the same major demanding requirements from the thin-client web application world.

- Merge the cuts of CGI program and web scripts together to create an easily developable, off-line parsing, optimizable, scalable, portable, and real-time overhead minimum server-side language.

- Specify the complexity of the web application easily and clearly to fit the 3-tier or multi-tier structure of the web applications naturally.

- Decompose the web application into irreducible reusable components, specify the communication or relation between different components and build the web application on top of these basic elements so that the developer could take all the advantages of layered modular approach including cooperation and code reusing.

While the traditional C++ language[15] seems an ideal candidate to achieve all the above requirements at first look, but in practice, it is too general for concrete web applications. Although some class libraries have been built, that general object-oriented language does not fit well in the specific problem domain like the web.

- It is a sophisticated programming language that is not good at specification.

- It lacks the ability to describe 3-tier application naturally.

- Even with the help of template concept, C++ is still not dynamic and nimble enough to easily develop tons of dynamic components fast.
All these shortcomings prohibit fast development of web applications using C++. But that does not mean that C++ is not general or flexible enough to develop web applications. On the contrary, at some level, as the most powerful programming language in practice, C++ is a perfect candidate for the server code just as NetObjects[16] has suggested. But the problem is: from here to another higher level where what developers need are not only delicate program structure and program performance, but also developing and deploying speed and component reuse, there is still a big gap. As pointed out sharply in [38], current web implementation model does not relate well to the state-of-art software development models. Web implementation is based on low-level technologies that do not provide high-level abstractions for sharing and reuse. The lack of suitable abstractions makes it difficult to construct frameworks that capture architectural design for rapid development and reuse in different parts of an application. However, if we can apply our experience and ad hoc knowledge about web application into this situation and develop a problem-oriented, or more specifically, web-oriented language that can transform the web problem directly into the C++ object model by a compiler, this gap could possibly be filled automatically so as to emancipate the speed of development that the whole e-commerce world is eagerly looking for.

Then comes the birth of PCR language.

PCR, which is an abbreviation of Page Connection Representation, is an intermediate language between 3GL and 4GL, mixing the specification style and programming style. This special style is created to meet all the special requirements listed before under the special environment called web.
Nevertheless, for the competing end, Java Servlets[18], Enterprise Java Beans[17], and UML[19] must be mentioned.

For the Java Servlet and Enterprise Java Beans, they developed similar problem domain concepts for the server side program. But the question is why all the server sites should go with one java way. It is not like Java Applet, which required universal runnable feature. For the server, performance and development speed instead are in the first place, not the universally executable feature. In PCR, the Page Connection Representation is an abstraction of the structure of the server module independent of the final intermediate code. The compiler can compile PCR into C++, Java, modular-3 and any other object-oriented languages to achieve the maximum performance on a specific platform. On the client side, PCR embraces all kinds of technologies. In some sense, PCR only gives an infrastructure for the server side web application and is pretty much implementation independent. Under this infrastructure, the web application could be developed quickly and managed easily while getting implementation sophistication leverage from client side technologies and powerful conventional programming languages.

For UML, its modeling philosophy is much like PCR. But UML is a general modeling language. It even specifies all the processes of a software engineering project. On the other side, PCR provides only one viewpoint of the web application. But beneath that viewpoint is a fully developed methodology. It pursues a balance between flexibility and controlliness. In other words, PCR is more web and server oriented.

Fig.1.1 shows the lifecycle of a web application[20]. PCR covers all the conceptualization, design, implementation and evolution processes. This places PCR in the most advanced
category of web development tools existing today: the model-driven generators. Like other Model-driven web generators, PCR provides the highest level of automation and lifecycle coverage among the above mentioned middlewares and web scripting systems by applying conceptual modeling and code generation techniques to the development of web applications. Only a few of other development tools exist in this category, namely HyperWave[21] and Oracle Designer/2000[22] in commercial world and Araneus[23], Autoweb[24], Strudel[25] and Web Architect[26] in research community. With its own unique Page Connection conceptual model, and the new multiple view and page decomposition concepts, PCR provides web architects with even more scalability, code-resuability and speed of development automation than its counterparts.

This paper is organized as follows. Chapter 2 introduces the serving mechanism and
architecture of the PCR generated server module. In Chapter 3, a description of PCR language itself is given in detail with sample PCR codes before a non-trivial example of PCR application EMMS is given in Chapter 4. Chapter 5 is devoted to a comparison of PCR with two representative web technologies: ASP and XML. The future expansion direction of PCR is discussed in Chapter 6 before leading to a final conclusion of the paper in Chapter 7. In appendix, a full specification of grammar rules for the business code of PCR and view of PCR are given, together with an introductory discussion of the compiler implementation. A more detailed description of the implementation of PCR compiler is beyond the scope of this paper.
Chapter 2

General Framework: a Top-down Approach

After a brief evolutionary review of the web serving mechanisms in this chapter, the general framework of PCR is introduced from a top-down approach.

2.1 Web serving mechanisms

Fig. 2.1 shows a global picture of 3-tier web applications. Between client and the web server, there is the web. For most of today’s application, it is in the layer of web services that attract most of the developers’ and site owner’s concerns. For the e-Commerce site owner, that is where the add-on value for its customer comes from. For the developers, that is the arena for all the server side technologies. In what follows, the major web serving mechanism will be listed and compared chronically.
Figure 2.1: 3-tier architecture of web applications
2.1.1 Static Page Serving

The static page serving mechanism is shown in Fig. 2.2. It represents the oldest way to serve pages using HTTP protocol [27] starting from NCSA httpd. They are simple and charming when they first come out to the public. But around 1996, with the exponential increase of e-Commerce market, they soon became too limited in their functionalities. In the new situations, lots of data need to be collected from database and get published through the net. Then the content of the page should be as dynamic as they are supposed to be in the e-Commerce world.

2.1.2 Dynamic Page Serving

With the demand for serving dynamic pages increases, the server side technologies use three routes to achieve this: one is CGI programs, one is web scripts and another is web server APIs. A CGI program can be implemented in any language and is very portable. But the
downside include process launching overhead and inability to manage and develop large-scale services. Web server API is very efficient in the sense of serving speed, but you have to know all the low-level APIs and program in a sophisticated (but powerful too) language like C. Even worse, every time when you add a module, the whole web server should be shut down to be recompiled or at least be added with a new dynamic library module. Today, the most favorable way to go is web scripts. These script languages will be embedded into the conventional HTML files. Every time before the web server sends back the requested file to the client, an interpretive module in the web server will interpret all the scripts embedded in the file and finish all the actions such as accessing a database or doing some logic computing. Finally, the furnished page with all the dynamic results in it will be send over to the client who could appreciate the dynamic content without knowing the backend scripting mechanisms. This process is demonstrated in Fig.2.3. ASP (Active Server Page) from Microsoft is a big success after it comes out, so is Cold Fusion from Allaire. In the freeware market, PHP and embedded Perl are coming silently but strongly. So does Server Side Javascript from Netscape. The cuts of these technologies are easy to write, very scalable and intermediately efficient. But with the increasing complexity of the web applications and increasing demands for the capacity of the services, the downside of these scripts are becoming more and more obvious. They are not in a layered component structure and does not describe the multi-tier architecture naturally, so does not fit the scenario of multi-tier web application. Although they have some escapes to external resources like DCOM objects[28] or CORBA services[29], the overhead and programming complexity involved in these still not fully accepted and wide-spread standards prohibit programmers from
deploying it in real practice. Also, all the escapes create some dependencies on external services. These dependencies do no good for a server whose ultimate goal is to be self-contained and autonomic. The most frequent question an application service provider will ask is: “How can I easily integrate and develop an efficient full-blown service site with all the modules install on my server fully under my own control?”

2.1.3 Dynamics of HTTP Transaction for PCR generated application server

The answer for the question in the end of last subsection is the PCR generated application server. As shown in Fig.2.4, PCR compiler will convert the PCR code, which describes the architecture of the service application and the function of each module, into a high-level object-oriented language like C++, and then use a third party compiler to compile this intermediate code into a final executable server module. This server module usually is an
Figure 2.4: Dynamics of HTTP Transaction for PCR generated Application Server
independent executable module, which can be optimized or even in multi-threaded mode. The communications between web server and this add-on giant can be in CGI if what you want is simpleness and clearness not super performance, and can be in FastCgi, a protocol that is originated from Open Market in 1996 to provide a fast, open, and secure Web server interface that solves the performance problems inherent in CGI, without introducing the overhead and complexity of proprietary APIs[32]. In this mode, the generated application server is a running program without the overhead of CGI and may maintain a persistent connection to a local database engine. The advantages of PCR generated service module is then obvious from the above discussion. It eliminates the on-line parsing needed by web scripts. Also, because the intermediate code is in C++, the final module can be optimized in different levels which is impossible for an interpretive web script. Besides the super performance of the generated service module, the real power lies in the PCR code itself. As mentioned in the end of the chapter 1, PCR combined the benefits of specification and scripting into a single language. This makes PCR capable of modeling a complex web application while still can program the service code for each module using script-like statements. Programming in PCR is like using UML for the architecture and rules specification, and Perl for action codes.

2.2 Page Connection Representation of a Web Application

For a programmer, the most important feature of PCR is the application structure awareness. PCR enforces its own view of web applications to programmers so as to ease the planning, modeling and deploying. The full methodology enforced by PCR also prohibits
common programming errors such as memory leaks that can cause serious problems for a running server.

In PCR, web applications are viewed as a group of coherent page objects. Each page takes care of a special task and fulfill its input/output through its web interface called views. It is the cooperation of this group of pages that realize the ultimate service functions of the web application.

The balance of autonomy and cooperation makes the web application a distributed synergetic application with a share storage space of database for all its active elements, pages (Fig.2.5). In this sense, all these pages are the actors or players on the stage of the local web server and database engine[33]. Web application is then the opera presented by
these actors to the audience over the web. In order to make the opera make sense to the audience, each actor, I mean “page” in this context, should play by some predefined rules. That is what PCR wants to specify. PCR associates a set of rules for each page. For each rule, PCR specifies the behavior of page when that rule is triggered. In that behavior part, the page object can communicate with other page objects, show the views of other page object and transfer the service requests to other page object. All these behavior induce the correlations between pages. These correlations should be rational, logical and making sense for the application. It is these correlations that make the web application an opera not a soprano. It is these correlations that make the cooperation possible. In this scenario, the number of actors can be increased to accommodate the need of application so as to direct a large scale complex application. Now one can understand the meaning of Page Connection Representation. Besides Fig.2.5, Fig.2.6 shows another scheme for a group of cooperating page objects, which will be the major metaphor and architecture skeleton for our PCR generated service module.

2.3 Architecture of the Page Application Object

The architecture and function of the single page object is important because they constitute the basic active element in our PCR scenario. As shown in Fig.2.7, each page consists of three parts or three tiers. First is view, which is the interface between user and the page object over the web. Second is the BLU(Business Logic Unit), which accepts input from the view, analyses it, has access to database and communicates with other pages. All the service code lies in BLU, which is the brain of each actor(page). Third is the database
Figure 2.6: The PCR's Pointview of Web Application: a group of cooperating page objects
image of this page. In fact, all the actors share the same stage, the same database engine.

The domain of the data each page responsible for might be different. Each page only takes care of a session of data in database engine. Database Image of each page can be defined as all the possible data in the database engine that will possibly be accessed by the page object. In future version of PCR, synchronization will be considered for the pages whose database images intersect with others'. One advantage of this layered structure is that each different tier can be developed independently and changed separately, if you keep the interface between them unchanged.
2.3.1 View

View for each page object is like clothe for each actor. It is what users see and input. Each view provides the page object a way to display the content of BLU. The separation of view and BLU is a good advantage over normal web scripts which embed the service code into the HTML file directly. In PCR, single BLU can have multiple views, each view can be selected dynamically. Multiple views will be covered in Chapter 3.

A view in PCR is a mixture of all client technologies modified by a set of PCR dynamics tags (illustrated in Fig. 2.8). PCR specifies a separate set of grammar for the views. Similar to XML, most of these dynamic tags represent some fields or members of the data area of BLU. Beyond that, view can also have access to methods of BLU, use control flow structure and manipulate temporary variables to program a sophisticated and delicate view for each object. Given all these freedoms, PCR view integrates the power of all current client technologies and the server side dynamic functionalities of PCR.

2.3.2 BLU

BLU is the central processing part for each page object. The architecture of BLU is shown in Fig. 2.9. Each BLU is divided into data area and behavior area. In data area, it contains all the data structures that will be used in processing. Some new data types has been invented to ease the View/BLU communications. The behavior area is composed by a dispatching unit (DU), a set of handlers and methods. Methods are just auxiliary member functions of the page objects like member functions in C++ or Java class. Handlers are the major service code providers. PCR associates each handler a rule (a logical condition) and
an action(service code). The dispatching unit(DU) of BLU will dispatch the user request from the view to the first handler whose rule gets satisfied. After triggering the action of that rule, the service code in action determines what to reply, calls service of other page objects or dispatches the request to other page objects.

In order to differentiate from the view, PCR has a set of separate grammar rules for the BLU. In the action part where all the service statements reside, all the PCR codes take a form of C language mixed with some features of Perl for string manipulation. This special style has been mentioned in Chapter 1, and will be more revealed in Chapter 3.
Figure 2.9: The Business Logic Unit
2.3.3 High Level Database Access Interface

The whole database structure of the web application will be specified in a "db" file. Its grammar will be given in Section 3.9. For BLU to have access to its database image, PCR provides a set of high-level database access interfacing methods for the database object. These methods take the form of operation, operant style. The operations can be common database access method like add, delete, update, retrieve and search. Operants can be tables in database and some PCR predefined data types, such as LIST, to be receiver or inputer to hold the actual data in BLU. This way, PCR avoids SQL query language by assigning each database operation and its operant a specific format. All proprietary database APIs and the type conversion needed afterwards are avoided too. PCR's philosophy is: use my data type, follow my rule and you will be ready to go!

2.4 Decompose the Page into Irreducible Components

In PCR's pointview of the web application, one page can be composed by many component pages. For a web architect, when designing the structure of the web site, what he has to do first is to decompose the pages into irreducible components. Great code efficiency can be achieved through the reuse of the same component in different pages. This component-oriented nature also enforces the programmer to think in a modular way. As illustrated in Fig.2.10, a product catalogue page can be decomposed into a head component, a location component, a list component and a footnote component. Obviously, the head component, location component and footnote component can appear in every page of that web site,
Figure 2.10: Decompose a sample product catalogue page into smaller components
whereas the list component may also appear in other pages too.

In this sense, the art of designing a large scale web application is the art of abstraction and decomposition. Once the active components have been recognized, the behavior and correlation (including hierarchy relations) can be described by the PCR language.

As time goes on, the component library of PCR will be established. In that library, all kinds of gadgets or components for different business models can be found. Building the complex web application will be a matter of choosing different building blocks and assembling them together using PCR.
Chapter 3

Description of the PCR Language

Let us begin with a quick introduction to PCR. Our aim is to show the essential elements of the language in real programs without delving into dazzling details. More exact specification will be given in later chapters.

PCR is the abbreviation for “Page Connection Representation”. As the name suggests, this is a programming language to describe views, inner structures and operational connections between “pages”, the main objects underlying the complex web applications. Since the structure of PCR reflects the basic 3-tier architecture of most web applications and the interrelations between objects in each layer, PCR is taking the traditional role of documentation for a big web project in some sense. The documentational style makes PCR code easy to read, write and maintain. Users can thus focus on the design of the architecture of the complex web applications. So in other sense, PCR is more web-oriented than general functional languages.

PCR normally will be first compiled into an intermediate high-level language like C++,
and then compiled into web server extended modules like cgi programs or an executable
multi-threaded server in fastcgi for better performance.

3.1 Getting Started

The only way to learn a new programming language is by writing programs in it. Let us
start by the universal first program "HelloWorld".

In PCR, currently we do not support functions like "print" yet. Since the view of each
page object is on the web over the client side (like their browsers), if we want to show some
words like "Hello World!", we have to make it appear on the HTML file transfered to the
client. In other words, PCR is dealing with a client/server situation behind the scene, but
hiding all the details by treating the view of the page and the logic of the page as the way
done in MFC( Microsoft Foundation Class)'s dialogs.

Here is the program called HelloWorld.pcr:

```plaintext
CPAGE HelloWorld {

STRUCT msgwin {

char msg[256];
}

HANDLER() {

msgwin.msg = "Hello Wolrd!";

GOTO THIS;
}
```
and its VIEW HelloWorld.htm:

```
<html>
<body>
<h1> <!DTAG=msgwim_msg> </h1>
</body>
</html>
```

Just how to run this program depends on the option you use to compile it into different target modules. Currently, we only supports cgi target module. So just type:

```
cgigen HelloWorld.pcr
```

at Unix shell. “cgigen” is the name of the current PCR compiler for generating cgi target modules. Then, a directory named "HelloWorld" will be generated. Inside that directory, there is a HelloWorld.cgi program. By invoking HelloWorld.cgi through normal hyperlinks in HTML files, or directly typing the URL of HelloWorld.cgi in the web browser, the following document will appear in the browser:

```
Hello World!
```

This program is composed by two parts in two separate files. One is HelloWorld.pcr, which is the second tier to describe the business logic for each page and connections between pages. For this simple program, there is only one page object “HelloWorld”. The page connection representation file is always terminated with “.pcr”. The second file is
HelloWorld.htm, which is a view template file for the page “HelloWorld”. The view template file is always terminated with “.htm”. Actually, PCR is capable of describing 3-tier application with a 3rd database tier. That one is missing for this small example.

In the .pcr file, each page object is started with the keyword “CPAGE” followed by the page name, in this case “HelloWorld”. Each page object is composed by data area and handler area. For HelloWorld, it only has one data member: STRUCT msgwin. Generally, for STRUCT data member, it can has arbitrary number of fields. Currently, PCR supports three basic types for fields: int, double, char array. Here we declare the msg as a character array using the same syntax as ANSI C. But unlike C, in PCR we don’t support pointer. So data declaration like “char * msg” is invalid.

The handler part of HelloWorld page only contains one handler. In principle, each page can have any number of handlers. The syntax for each handler definition is more like a C++ class’s in-line member function definition. There is no such thing as function declaration in PCR. The name of this handler function is always the keyword HANDLER. The traditional parameter list area contains the event to trigger this handler. In most GUI programming languages like java, MFC and X/Motif, handlers or callbacks are event driven and they all have specified events for different class. But here in PCR, the handlers are user-input driven, or user-data driven. So each event is represented by a logical expression consisting of the datum from page’s data area. This kind of event is programmer-defined, so is more flexible and more business-oriented. Keep in mind, in most web applications, which handler to choose is determined by what the user input in a html form. But in this simple example, the missing of events means that the only handler will always be triggered
when the user calls this page's server module.

Be careful, inside each CPAGE object are the service data and service logic for the user request. So each page object is activated per request. There will be a request dispatching mechanism in the final compiled server module, but that part is unseen from the programmer's view. All the service code should enter the handlers and there is no such top-level entry point like "main" in PCR.

Inside the handler are just script style statements used to deal with the user request. Here we just make the msgwin.msg to store the string "Hello World!". At the exiting point of the handler, the current page always has to determine who is the next page to go. It is here that comes in the connection between different page objects. But for this example, GOTO THIS means to show off the view of itself. Just one more words, the handler list for each page object can be missing and that page's server module will do nothing when a request comes in; or it can has many handlers and the event of the handlers coming first on the handler list are checked first. Once the logical expression for one of the events is checked to be TRUE, that handler is triggered, and later handlers will not be checked anymore.

HelloWorld.htm is a normal HTML file with an embedded dynamic tag \( \texttt{!DTAG = msgwin.msg} \). This dynamic tag is used to display the dynamic content of the msg field for the msgwin data member of the page "HelloWorld". The "_" is used for "msgwin.msg" instead of ".". This is a convention for template file. We only use "_" in template file because we try to keep the identifier like "msgwin.msg" still a valid variable name to get the input from the input field of a html form. We will cover the input part later.

Interestingly, PCR only deals with the dynamic part of the view of the page object.
The dynamic tags embedded is more like a macro in C, but the content of that macro is
dynamic in a sense that it depends on the real value of the data member inside each page
object. As to the concrete modeling language used to make up the view template, PCR
leaves the door wide open to all the available and upcoming client technologies. You can
mix the DTAG with HTML, CSS, XML, JAVASCRIPT, VBSCRIPT, just any kind of text
based client tools. PCR makes this happen because PCR only put its feet on server side.

3.2 Structure of PCR Program

From the last section, we already have seen the basic structure of PCR program. More
generally, any PCR program contains a core file postfixed by “.pcr”. That is the business
file or the page file. In the page file, there is usually a list of CPAGE object definitions.

```
CPAGE  page_name  {

    page_body

}

CPAGE  page_name  {

    page_body

}

...  
```

For each page object, it follows the following grammar summarized from the discussion
of last section:

\[
\text{CPAGE ~page\_name} \{ \\
\text{data\_area} \\
\text{init} \\
\text{handlers} \\
\text{methods} \\
\}
\]

\text{init} method is invoked for each page object when the request coming in, so as to do the initialization work for the page service. The grammar for \text{init} is

\[
\text{INIT} () \{ \\
\text{service\_code} \\
\}
\]

The \text{init} method can be absent. If that is the case, initializations will not be carried out.

For each handler,

\[
\text{HANDLER (event)} \{ \\
\text{service\_code} \\
\}
\]

where

\[
\text{event : logical\_expression}
\]
Figure 3.1: The Correspondence between PCR source modules and target modules

```
 service_code : statements
```

As mentioned in the last section, each page object has at least one view, which is specified in the view template with the same file name as the page name and terminated by “.htm”. Multiple views of one page object is allowed in PCR and it will be covered in section 3.12.

Fig.3.1 shows the correspondence between PCR source modules and target modules, and the function of PCR compiler. Also as shown in the figure, programmers have to supply PCR compiler with a database specification file terminated with “.db” if the program has a database tier. The database specification file will be covered in section 3.12.
3.3 Basic Types, Dynamic ADT, Static ADT and Reference Type

The data area of each page object is composed by the definition of page level data objects with different types. This level of objects is classified into three types: basic type, dynamic ADT, static ADT and reference type. The current supported basic type is INT, dynamic ADTs are STRUCT, IN_STRUCT, LIST and reference type is REF and DB. Static ADTs have no keyword in PER, they are user defined and generated by the dynamic ADTs.

There is not much to talk about basic types. Now we only support one type: INT at page data area level. This is mostly used in loop control in the service code. More traditional types will be added if necessary.

STRUCT, IN_STRUCT and LIST are dynamic ADTs. By “dynamic” we mean these types are more like class template in C++. Static ADT is more like a traditional class or a simple type. But it only comes in by means of dynamic ADT in PCR. Not like C++, which differentiates the class and object concept clearly, in PCR, the definition of the static ADT and its first copy of object are taking place at the same time. So when you define “test” through a PCR supported dynamic ADT STRUCT as:

```c
STRUCT test {
    int data;
}
```

A new user-defined static ADT called test is created, as well as the first instantiated copy
of this ADT: test. The newly created static ADT test is useful because later we can define
reference object in data area like:

    REF test rec;

Later in the service code of the page object, reference variable rec can be used on the left
hand side of the assignment statement to refer to the internal data object returned by the
member function of another user-defined static ADT. For example,

    rec = Listarticle.GetNext();

Listarticle is an object of the user-defined static ADT Listarticle through the LIST dy-
namic ADT, GetNext is its member function and it returns an object of test type. Once
the assignment has been made, the members of the object which rec refers to can be ac-
cessed through the normal way like “rec.data”, assuming data is that object’s member.
For the robustness of the memory management of the target server module, PCR does not
support pointers. In order to avoid the time-consuming object copy while still making the
program robust, PCR borrowed the reference concept from C++ and Java. Most of the
PCR supported dynamic ADT’s member function will return reference to some internal
allocated object. As different from a pointer, a reference variable in PCR can only be
used on the left hand side of the assignment statement and can not be manipulated in any
expression like in C or C++.

STRUCT dynamic ADT has been covered in the “HelloWorld” example in the last
IN_STRUCT objects have a very similar role as STRUCT, and the only difference is that each field of IN_STRUCT object has a correspondent input variable in the view template of that page. Before the program flow enters the handler, the fields of the IN_STRUCT object already has been instantiated with the data retrieved from the HTML form. This data transmission may be carried in CGI protocol, but the user don’t have to know it. PCR will take care of it, since PCR is network transparent. In other words, once you enter one of the page object’s handlers, you can freely manipulate the data in the IN_STRUCT object as they represent the data users input from the view template. For example:

```c
echo.pcr:

CPAGE echo {

    IN_STRUCT flag {
        int     firsttime;

    }

    IN_STRUCT content {
        char     msg[64];

    }

    HANDLER(flag.firsttime==1) {
        content.msg = "First time shown, no message to echo yet!";

        GOTO THIS;
    }

    HANDLER(flag.firsttime==0) {
```
GOTO THIS;

}

}

}

}


echo.htm:

<html>
<body>
<h1> echo the message you input </h1>
<h2> ECHO: <!DTAG=content_msg> </h2>
<form method=post action=echo.cgi>
<input type=text name=content_msg>
<input type=hidden name=flag_firsttime value='0'>
</form>
</body>
</html>

When you try to bring up the view of the page for the first time, you can use the hyperlink like:

<a href='echo.cgi?flag_firsttime=1'></a>

The above "echo" page is to echo the message you input into the text input field of the form in the view of echo. An additional
IN_STRUCT flag {
    int firsttime;
}

is defined to differentiate whether the page is activated first time or not.

There is also a special reference type called reference to database.

DB magazine;

defines a special reference magazine, refering to a database with the exact same name: 
magazine. This is a more static reference than the normal reference variable because magazine then can not be assigned to refer to a new database. Later expansion of this database 
reference concept will be carried out if necessary.

3.4 LIST

LIST is a dynamic ADT in PCR. It is a very important component supported by PCR. It is usually used as a container to store the results retrieved from the database. Afterwards, service code or views can have access to the records in LIST through well-defined interface methods of LIST. In the following example:

LIST Listarticle {
    int id;
    char title[64];
Listarticle is the name of the static ADT generated by the dynamic ADT LIST. Listarticle is a list ADT whose records contain the fields listed in the definition. Besides these records, LIST will also generate core methods for Listarticle. LIST also generates a static ADT named ListarticleRec in the example. That is the type for the record stored by Listarticle. PCR automatically generates this static ADT for user because later some Listarticle member function like GetNext() will return the reference to the record and that reference is of the type REF ListarticleRec. The most frequently used core methods of LIST objects are:

void Reset() Reset the traversal entry point to the head of the list

int GetNumber() return total number of the records in the list.

REF ListarticleRec GetNext() return the reference of the next record in the list and move the index pointer one record forward.

3.5 Variables and Arithmetic Expressions

All the defined data objects and their data members can be used as variables. And the arithmetic expressions containing the operation of + - x ÷ % are supported by PCR. The expression can be assigned to a variable in an assignment statement.

But currently, the arithmetic expression is not supported in the views of PCR page object. That is, in the DTAG environment, you can only make simple assignment between
variables, or constant to variable. But there is no expression on the right hand side of a statement. For example,

\[ \langle!DTAG \ a = b\rangle \]

is valid if \(a\) and \(b\) are both defined variable in the corresponding CPAGE, while

\[ \langle!DTAG \ a = a + b\rangle \]

is invalid, because there can be no expression like \(a + b\) in DTAG environment in view template. The reason PCR disabled the use of expression in view template is to restrict the function of the view is only to show the data inside CPAGE object. All the complex operations should be carried out in page objects service code or encapsulate the complex function in the methods of components like LIST. Users can only have access to the internal workings of complex components through the methods interface. This restriction separates the functions of view and business logic clearly and increases the robustness of the program.

There is a class of special variables: strings. In this version of PCR, they can be members of a STRUCT with fixed maximum length or string constant like "Iamastring". Space is allowed inside a string. Strings can be assigned to strings directly, as

\[ a.b = "string"; \]

or catenate two strings together by
a.b = a.b + "string";

Additional features of the string operation will be added on in later versions of PCR, including substring matching and regular expressions. PCR will cover all the powerful features of script languages like Perl.

3.6 Statement

Statements constitute the main body of the service codes in page handlers. They are classified into following categories:

assign_statement

if_statement

ifelse_statement

elseif_statement

or_statement

while_statement

do_statement

new_statement

update_statement
Some kinds of the statements have been used in previous examples. In this section, only three of them will be introduced, the others will be covered in later sections.

\[ \text{assign statement} : \text{variable} = \text{expression} \]

\[ | \text{variable} = \text{method} \]

\[ \text{escape} : \%\{\text{code in any language}\}\% \]

\textit{escape} gives users a chance to escape PCR temporary to any other programming language. PCR compiler directly copies any text character between "\%{" and "}\%" into the generated intermediate code. PCR is mainly a script style language. Sometimes it may lack the special features which can only be realized in the intermediate code like C++ which PCR supports right now. So \textit{escape} feature makes the C++ tools always available to more sophisticated users.

\section*{3.7 Flow Control}

\subsection*{3.7.1 Logical Expression/Condition}

Currently, logical expressions in PCR are in the following formats:

\[ \text{variable logic opr variable} \]
And the `logic_opr` can be `<=`, `>`, `>=`, `==`, `!=` where `==` and `!=` can also be used for strings. Logical expressions can be used in the flow controls in the service code, or to represent the events to trigger the handler as described in section 3.2.

In the view template, in order to avoid collision with "<" and ">" symbols in DTag, the `logic_opr` instead represents for `ST SE GT GE EQ NE`.

### 3.7.2 IF Statement

IF construct can be in anyone of the following formats:

```plaintext
IF (logical_expression) {
    statements
}

IF (logical_expression) {
    statements
}
ELSE {
    statements
}

IF (logical_expression) {
    statements
}
```
ELSE IF (logical_expression) {

    statements

}
ELSE {

    statements

}

In the view template, all the flow control constructs and statements will appear in the

(!DTAG) environment like follows with a slight different grammar:

    <!IF logical_expression >

    <!DTAG statements >

    <!ENDIF >

    <!IF logical_expression >

    <!DTAG statements >

    <!ENDIF >

    <!ELSE >

    <!DTAG statements >

    <!ENDELS >
Apart from the `<!TAG>` environment and slightly different construct for ENDIF, etc., the logical expression and statements are in the same grammar and use the same data objects as the service code in CPAGE.

The explanation of the execution sequence altered by the above language construct is trivial. It is very similar to C language.

### 3.7.3 Loops — FOR

For loops are allowed in PCR, but in a simpler form than in C language,

```c
FOR (variable = expression TO expression) {
    statements
}
```
In the view template,

```xml
<!FOR (variable = method.variable TO method.variable >

<!DTAG statements >

<!ENDIF >
```

### 3.7.4 Loops — WHILE

```xml
WHILE (logical_expression) {

 statements

}
```

In the view template,

```xml
<!WHILE logical_expression >

<!DTAG statements >

<!ENDWHILE >
```

### 3.7.5 Loops — DO-WHILE

```xml
DO {

 statements

} WHILE (logical_expression);
```
As an exception with other loop constructs, DO-WHILE loop has not been supported by PCR yet. It will be supported in the later version of PCR.

3.8 Method

PCR supports methods or member functions for all the objects, but PCR does not encourage the user to program the component apart from CPAGE. In PCR's philosophy, the programmers of PCR only use the components supported internally by PCR or externally by contributors who comply with the PCR’s dynamic library protocol. The only place the method can enter in PCR code is at the CPAGE level. That is, users can define page objects’ inline member functions, which are called methods here in PCR. We stick to this rule, because we try to emphasize the script style of PCR and its easy-to-use philosophy.

\[
\text{method\_declr} : \text{VOID \ method\_name\(\text{arg\_list}\)}
\]

\[
| \text{return\_type \ method\_name\(\text{arg\_list}\)}
\]

The \text{arg\_list} is type-argv pairs just as in C. So the concrete explanation of the methods is trivial. They resume the same style of C.

The methods of objects or components supported by PCR are important, since in many cases they provide the only interface to exploit the functionality of these objects. The most significant component, or dynamic ADT supported by PCR currently is LIST. The methods of the static ADT generated by LIST have been introduced in section 3.4.
3.9 Database Specification File

Database Specification File always has a suffix "db", it specifies the structure of the underlying database that the BLU will have access to. It should be in a 3NF deduced from a E-R model[34, 35]. Once the PCR compiler gets the information of the database image, high-level database connectivities are made possible by hiding all the underlying implementation details using proprietary APIs[13]. The database access methods are introduced in the next section and the current implementation of PCR compiler covers MySql DBMS.

The Database class can be defined as follows:

```plaintext
dATABASE   db.classname
{
    TABLE   table.name   {
        type   data   create_definition
        ...
    }
    ...
}
```

Once a class has been defined, a database object could be declared as

```plaintext
db.classname   dbname
```

Now, the database object `dbname` has all the dynamic methods that will be covered in the next section. This object can be referenced in BLU.
3.10 Database Connectivity

Because PCR deals with 3-tier web applications, database connectivity is an important part of PCR. One of the main purposes of the creation of PCR is to make the connection between the components like LIST in business logic more natural with database APIs so that PCR relieves the programmer from the burden to learn all the APIs for different types of databases. The dynamic ADT DB has five methods supported by PCR currently. And the current supported database engine is MySQL. Things that have to be mention is that these methods are dynamic methods for the dynamic ADT DB. By "dynamic" PCR means these methods are more like macros or function templates. The types of their arguments are mixed with dynamic ADT and static ADT. For those with dynamic ADT, all objects from a set of static ADT can fit in as arguments, so long as these static ADT belong to the dynamic ADT in the deceleration of that method.

**insert** `dbobj_name.insert(table_name, struct_obj)` Insert the `struct_obj` into the table with `table_name` in backend database. PCR makes an assumption that both `table_name` and `struct_obj` have a field called "id" and "id" is the primary key for that table. This key is auto-incremental. The insert returns OK (0) on success, ERROR_SELECT_DB if it cannot select the correct database and ERROR_QUERY if the sql query failed.

**update** `dbobj_name.update(table_name, struct_obj, key-field_name)` Update the row of the table with `table_name` with the content of the `struct_obj`. The key used to select the rows is by `key-field_name`. Return value is the same as insert.

**delete** `dbobj_name.delete(table_name, struct_name, key-field_name)` Delete the row of the
table with `table_name` with the content of the `struct_obj`. The key used to select the rows is by `key_field_name`. Return value is the same as insert.

**retrieve** `dbobj.name.retrieve(listname, sql_query)` Retrieve all the rows of data using the query string `sql_query`. The results will be stored into the list object of `listname`. According to the STRUCT type of the records in the list, retrieve will do the automatic type transformation for you from the plain retrieve result of the sql query. But in order to promise the correctness, the field types in the STRUCT record should match those retrieved from the `sql_query`. `sql_query` can be a constant string or a string variable. It can be any kind of sql query to retrieve the data from relational database. PCR opens this opportunity to give the user the freedom to write complex queries when needed, like the join of two tables or embedded sql in Oracle. However, the trade-off is that it is the responsibility of programmer to promise the correctness of the results in the final container `listname`. PCR does not parse the `sql_query`. Return value is the same as insert.

**retrieve** `dbobj.name.retrieve(struct_obj, sql_query)` The format of this retrieve is exactly the same with the last retrieve from the surface. The only difference is that the first argument is a `struct_obj` instead of a LIST object. This method is used to retrieve only the first record of the query result or the only one record. The calling of this or the last retrieve method is exactly the same and PCR will search through the program's symbol table to find out whether the first argument is STRUCT object or LIST object. Then it will generate the proper code for the user.
**search** 

dbobj.name.search(listname, sql_query, offset, rows, total_rows) Search the database using sql_query. The essence of the search method is basically the same as the retrieve.

The difference is PCR allows programmer to pass in three more variables. Since in many cases the search results can not be shown all at one time in one page, one call of the search method only returns the rows number of rows starting from the offset in the total matched rows. total_rows is filled in by the search methods to tell the user the total number of rows in the search match the sql_query. Programmer can call several times of search with different offset to show all the results. Search returns the normal return values of database methods.

Summarily, the advantage of database methods in PCR is that a programmer does not have to know the database APIs of underlying database engine. If you want to insert, update or delete data from a table, put your data in a STRUCT container and just do it. If you want to retrieve a single or multiple record of data from database, prepare a proper STRUCT container or LIST container, make up an correct sql-query string and do it! All the data will be in the STRUCT or LIST, ready to be used. No type transformation is needed.

### 3.11 Multiple Pages and Page Connections

PCR is created to describe the complex relations between multiple page objects. Below is a simple example program of two strongly coupled page objects.

```python
switch.pcr :
```
CPAGE page1 {

IMPORT page2;

EXTERN page2 a;

STRUCT flag {
    int firsttime;
}

STRUCT msgwin {
    char msg[64];
}

HANDLER(flag.firsttime==1) {
    msgwin.msg = " ";
    GOTO THIS;
}

HANDLER(flag.firsttime==0) {
    PAGE a = NEW page2;
    a_msgwin.msg = msgwin.msg;
    UPDATE a;
    GOTO a;
}
}

CPAGE page2 {

IMPORT page1;
EXTERN page1 a;

STRUCT flag {
    int firsttime;
}

STRUCT msgwin {
    char msg[64];
}

HANDLER(flag.firsttime==1) {
    msgwin.msg = " ";
    GOTO THIS;
}

HANDLER(flag.firsttime==0) {
    PAGE a = NEW page1;
    a_msgwin.msg = msgwin.msg;
    UPDATE a;
    GOTO a;
}

page1.htm:
   <html>
   <body>
Once you type into the input text field of one page and click submit button, that message will appear on top of another page and then can switch back and forth between these two pages. The identity of the page is identified by the title on the top of each page. Of course, this program is trivial, but it reveals the most important property of PCR: it can describe the connection between multiple coherent pages! These pages with connections
between them constitute a graph. When the user sits on one page, the next page to go is determined by the user inputs and the business logic to process them. So this graph is pretty much like a state-machine. It is this state-machine that sits behind the scene of the complex web application, which might be an e-commerce site or a news-media site. And it is the duty of PCR to help specify this state-machine clearly and efficiently. Fig. 3.2 is a simple illustration of the state-machine.

3.12 Multiple Views

As shown in Fig. 2.7, each page object in PCR program can have multiple views. All the views correspond to the same data structure in business logic part. Different views just use different ways to show out the page data. For example:

```cpp
CPAGE TestPage { 
  VIEW v1;
}
In the above page, there are two views. In addition of the default view defined in TestPage.htm, it has another view: defined in v1.htm. If the value of logical_expression is non-zero, TestPage will show the view v1 instead of its default view.
Chapter 4

Application of PCR to E-Magazine Management System (EMMS)

As an example of a non-trivial application of PCR, the infrastructure of a EMMS (E-Magazine Management System) is introduced in this chapter. And the PCR code skeleton is described to demonstrate how PCR can be used to specify the infrastructure and relations between different elements of EMMS. The working example is the administration system for edge e-magazine[30], which use Apache[31] as the web server and MySql[13] as the database engine.

4.1 Database Image of EMMS

Fig. 4.1 shows the image of EMMS, which is basically the E-R model of the database in a relational database management system like MySql or Oracle. It is already in its 3NF
Figure 4.1: Database Image of EMMS

Each entity is basic object in the whole EMMS. Besides the data stored in each object, there are methods associated with them, which furnish the Add, List and Update functions. PCR codes, which lie on top of the database, provide EMMS with a coherent web interface between the user and the database. Furthermore, it specifies the rules of the access of these methods based on the user input. In this sense, Pages in PCR code are actors who play on the stage of the database with the rules specified by the Handlers associated with each Page object. All these active Pages serve the middleware between the user and the database over the web. In order to let the PCR compiler know the structure of database image, EMMS also has to supply it with a database specification file: “mag.db”:

```
DATABASE CMagazine
```
TABLE account {
    int id;  <primary_key>
    char[32] username;
    char[32] passwd;
    int issueID; <foreign_key>
}

TABLE article {
    int id;  <primary_key>
    char[128] title;
    char[128] keyword;
    int issueID;  <foreign_key>
    int columnID; <foreign_key>
    int authorID;  <foreign_key>
    int ord;
}

TABLE issue {
    int id;  <primary_key>
    int number;
    char[64] editor;
}

TABLE mycolumn {
int id; <primary_key>
char[64] name;
char[64] host;
}

TABLE author {
   int id; <primary_key>
   char[64] name;
   char[128] affiliation;
   char[64] email;
   char[64] tele;
}

CMagazine magazine;

In mag.db, CMagazine specifies the table structure of the database template. And the final line defines “magazine” as an database object from that template.

4.2 Page Connection Representation of EMMS

As illustrated in Fig.4.2, the whole middleware of EMMS can be represented by a group of coherent pages. There are relations between pages. They can be GOTO statement in PCR or normal hyperlinks in html. The current page is the page that is being shown on
Figure 4.2: Page Connection Representation of EMMS
the user's web browser. Depending on the user input or the calling page, the current page business logic on the server side will determine which next page to go based on the rules contained in the handlers. This reflects the computing essence of the middleware between the user and the database over the web: it is an extended automata whose nodes share the same storage space (database).

4.3 Illustrated PCR Code Skeleton for EMMS

The following PCR code segment is from "mag.pcr" file. It is used here to illustrate the page connection relation between Login page, Admin page and ListColumn page. From Fig. 4.2, one can easily extend this code to specify the behavior and relation of other pages as what their names imply.

```
CPAGE Login{

  IMPORT Admin;

  EXTERN Admin a;

  INT ret;
  INT i;

  DB magazine;

  IN_STRUCT account {
```
int id;
char username[32];
char passwd[32];
int issueID;

}  

STRUCT issue {
    int id;
    int number;
    char editor[64];
}

STRUCT query {
    char command[256];
}

STRUCT errmsg {
    char msg[128];
}

HANDLER() {
    /* retrieve the account record */

    query.command = "select id, username, passwd, issueID from account where username='" + $(account.username) + '" and passwd ="' + $(account.passwd) + '";

    ret = magazine.retrieve(account,query.command);
IF (ret==0) {
    PAGE a = NEW Admin;
    /* verify passwd */
    a_msgwin.msg = "User " + $(account.username) + " Login in successful!";
    query.command = "select id, number, editor from issue where id=" + $(account.issueID);
    ret = magazine.retrieve(issue,query.command);
    IF (ret==0) {
        a_msgwin.msg = "Welcome " + $(account.username);
        a_session.id = account.id;
        a_session.username = account.username;
        a_session.issueID = account.issueID;
        a_session.issueNumber = issue.number;
    }
    ELSE {
        errmsg.msg = "cannot retrieve current issue number from the user account!";
        SHOW THIS;
    }
}

UPDATE a;
SHOW a;
ELSE {
    errmsg.msg = query.command;
    SHOW THIS;
}

CPAGE Admin {
    SESSION;
    STRUCT msgwin {
        char msg[256];
    }
}

CPAGE ListColumn {
    SESSION;
    INT ret,j,i;
    INT k;
    DB magazine;
    IN_STRUCT command{
        int del;
    }
LIST Listcolumn {
    int id;
    char name[64];
    char host[64];
}

LIST Listedit {
    int del;
}

LIST Listarticle {
    int id;
}

STRUCT query {
    char command[256];
}

STRUCT msgwin {
    char msg[256];
}

REF ListcolumnRec rec;

REF ListeditRec pe;

HANDLER(command.del==0) {
    /* retrieve all the column names in this issue */
ret = magazine.retrieve(Listcolumn,"select id,name,host from mycolumn");

msgwin.msg = " ";

UPDATE THIS;

SHOW THIS;

}

HANDLER(command.del==1) {

/* first del the checked column */

ret = magazine.retrieve(Listcolumn,"select id,name,host from mycolumn");

Listcolumn.Reset();
i = Listcolumn.GetNumber();
rec = Listcolumn.GetNext();

FOR (j=1 TO i) {

pe = Listedit.GetCgiInput();

IF (pe.del == 1) {

query.command = "select id from article where columnID="
+ $(rec.id);

ret = magazine.retrieve(Listarticle,query.command);

IF (ret != 0) {

msgwin.msg = "don't know whether the column have articles";

}
SHOW THIS;

}

k = Listarticle.GetNumber();

IF (k > 0) {

    msgwin.msg = "The column in is not empty, cannot be deleted";

    SHOW THIS;

}

ret = magazine.delete(mycolumn, rec, id);

rec = Listcolumn.Delete();

}

ELSE {

    rec = Listcolumn.GetNext();

}

}

msgwin.msg = " ";

UPDATE THIS;

SHOW THIS;

}
Chapter 5

Comparison of PCR with Related Web Technologies

A brief survey of current web application technologies has been given in the Introduction chapter. In this chapter, the two representative popular web technologies, ASP and XML, will be examined in detail with comparisons with PCR. A more detailed survey of web application technologies including researching projects are covered in [20]. With the understanding of PCR architecture in this paper, more comparisons with other existing systems could be made separately.

5.1 Active Server Page and PCR

From Chapter 1, the advantages of PCR over normal web scripts are made clear. Active Server Page (ASP) as a tool in web scripts category shares the same weakness:
- interpretive nature
- unoptimizable
- not incorporate 3-tier concept into web scripts
- poor portability

But with the advent of COM technology and the combinational use of ASP and COM[10, 11], Microsoft pretty much cured above weaknesses. COM is a component technology on the same level as CORBA and JavaBeans. It is the building block of Microsoft applications as its precursor ActiveX Control. When the company’s attention turned to the web, COM and its distributed counterpart, DCOM, are providing the whole server side infrastructure for building web applications on Microsoft platforms. Following is an illustrated code segment for the ColumnList Page Object for the EMMS written in ASP and COM. The first file contains the ColumnDataAccessor COM. It has a method called GetColumns.

Private Const strConnection As String = "Provider = SQLOLEDB;" & _
"Data Source=mag;"
Public Function GetColumns() As ADODB.Recordset
  Dim objConn As New ADODB.Connection
  Dim objRecColumns As New ADODB.Recordset

  objConn.Open strConnection
  objRecColumns.CursorLocation = adUseClient
  objRecColumns.Open "mycolumn", objConn, adOpenKeyset, _
Set objRecAuthors.ActiveConnection = Nothing

Set GetAuthors = objColumnss

Set objRecAuthors = Nothing

objConn.Close

Set objConn = Nothing

ColumnDataAccessor COM is programmed in VBScript. It sets up a connection to the database and uses ADO (ActiveX Data Objects) to retrieve data from the database. All the column records are then stored in objRecColumns, which is a Recordset object of ADO. After defined the above data-centric COM, an ASP script, ListColumn.asp, is further defined to display the data to the client:

<!-- line below references the ADO type library -->
<!-- METADATA TYPE="typelib"
FILE="C:\program files\common files\system\ado\msado15.dll"-->

<HTML>

<HEAD>

<TITLE>Creating a Disconnected Recordset on the Web Server</TITLE>

<STYLE TYPE="text/css">  ' a little formatting

BODY {font-family: Tahoma, Arial, sans-serif; font-size: 12px; font-weight: bold}

TD {font-family: Tahoma, Arial, sans-serif; font-size: 12px; font-weight: normal}
Dim objColumns
Dim objRS

Set objColumns = Server.CreateObject("ColumnDataAccessor")
' create the object
Set objRS = objColumns.GetColumns
Set objAuthors = Nothing

' display the contents of objRS
Response.Write "<TABLE><THEAD>" & _
    "<TH> NAME </TD>" & _
    "<TH> HOST </TD>" & _
    "</THEAD>"

While Not objRS.EOF
    Response.Write "<TR><TD>" & objRS("name") & "</TD> " & _
The above combination of ASP and COM is illustrated in Fig. 5.1. From there, one can see the correspondence of ASP with PCR view, COM with BLU. But in the case of PCR, it does not stop there. PCR furthermore provides a concrete, yet general enough, conceptual architecture for web applications. Under this viewpoint of page objects, the architecture of web applications is more unified. The whole application is first decomposed into pages. By tying the VIEW with BLU, PCR gives page objects more autonomy. Then, PCR puts more focus on the relation between these pages. Here come the major advantages of PCR over the combination of ASP and COM. Just as shown in Fig. 2.5 and Fig. 2.6, first, by appointing the major type of players of web application only to be page objects, PCR makes the application architecture flatter. This makes the PCR program easy to think and develop. Second, the main engine for each page is moved from VIEW/ASP layer to BLU, which corresponds to COM in MS platform. Making BLU as the center of command localizes the action part, which further separates the static part and dynamic part. After setting this background, the programming of PCR becomes just the weaving of page objects.
into a final service module.

In one word, the combination of ASP and plumbing of COM still lacks the support of conceptual architecture of general web problem domain. PCR noticed it and provides developers with more problem domain intimacy. Of course, Microsoft noticed it too and published MTS (Microsoft Transaction Server), ADSI (Active Directory Service Interfaces) and ATL (Active Template Library). MTS is essentially a component-based run-time environment that provides the infrastructure required for writing n-tier applications that are scalable, robust, transactional and secure. With the additional help of the directory services from ADSI and the component library ATL, Microsoft is providing a whole package for web application development with CASE tools. This package together with MSMQ (Microsoft Message Queue) and IIS Web Server sets up the Microsoft’s whole framework for developing
internet/intranet based applications with all the features they typically need. That newest framework is called Windows DNA (Distributed interNet Application Architecture) [11]. Yes, Windows DNA is awesome. It is awesomely powerful, but it is also awesome to learn. More importantly, it is a group of products coined by Microsoft to continue its monopoly dream that has been achieved successfully using Visual C++ on PC platforms in its old monopoly domain. But as the web emergent as a market for free new competitions and with the super performances of Freewares like Linux, FreeBSD, Apache, PHP and MySql, portability will be the major obstacle for the Windows DNA to control the major market share. PCR with major page connection enhancement over PHP web scripts surely still cannot compete with Windows DNA in full scale, but it tries to focus on its unique flat and self-organizing conceptual architecture features so as to be more light-weight, easy to learn and develop, and extremely portable. All these features together with the open source nature make PCR a hopeful candidate to set up the web application infrastructure for Freeware world.

5.2 XML and PCR

Another technology that makes such a big fuss in web community that makes it hard to be ignored when delivering a new web application technology is XML. Stimulated by the transition from traditional EDI (Electronic Data Interchange) to web-based B2B exchange process and the application of intelligent agents to the e-commerce world, XML is increasingly being targeted as the next-generation data representation for the web community [36, 37, 38, 39]. While PCR is targeted as a development tool to supply complex web applications with an general infrastructure, the relation of PCR with XML, and the position
of each tool in the whole picture of the web application infrastructure are worthwhile to address. In my vision, both PCR and XML specialized in its own relatively separate area of web application, which will be discussed in Section 5.2.1. But in a more grand scenario of cooperating web applications, XML can be fitted into the PCR model seamlessly and enable PCR generated service modules or page objects to cover a more general range of web application types including agents. The fitting of XML into PCR will be discussed in section 5.2.2.

5.2.1 XML as a Client Side Technology

XML is no more than a specification for data storage medium. It is a subset of the Standard Generalized Markup Language (SGML) defined in ISO standard 8879:1986. Not like HTML, which only provides one way to present the data, XML enables one to define his own markup language. And this markup language is not used to define the layout of the document but the structure of the document and logical meaning of its elements.

XML models the data in the following way. Each XML document is divided into a series of storage units, called entities. Each entity is composed by a list of elements. Entities in a XML document correspond to physical storage units, they can be files in local machine or somewhere over the network, database records and even pieces of memory. The elements inside each entity gives the entity its logical structure. Each element is a piece of information enclosed by begin and end tags. Tag refers to a specific text string used to represent an element. name is the name of element which starts with <name> and ends with </name>. The following is an example XML document for the ListColumn page object from the last
chapter.

<?xml version="1.0"?>
<!DOCTYPE ListColumn SYSTEM ListColumn.dtd[]>

<ListColumn>
  <!-- column element 1 -->
  <column>
    <id>1</id>
    <name>news</name>
    <host>alien</host>
  </column>

  <!-- column element 2 -->
  <column>
    <id>2</id>
    <name>reviews</name>
    <host>alien</host>
  </column>

  <!-- more column elements enter here -->
</ListColumn>
This example shows some additional XML markup components in XML 1.0:

- Processing instructions
- Document type declarations
- Comments

Since XML documents are designed for processing by XML processors, they must adhere to additional rigid syntax. These constraints on a XML document's layout and structure are given in the Document Type Definition (DTD), or Schema, which specify the relations between markup components. The following is the file "ListColumn.dtd" for the above XML document for ListColumn page object:

```xml
<!ELEMENT ListColumn (column)+>
<!ELEMENT column(name, host)>
<!ALLLIST column
del (1|0) "0">
<!ELEMENT name (#PCDATA)>
<!ELEMENT host (#PCDATA)>
```

DTD is very important in establishing whether the document is valid or just well-formed. A well-formed document is one that adheres to XML syntax rules, whereas a valid document is
a well-formed document that also adheres to a DTD. A quick comparison of XML document and its DTD with the components in PCR's 3-tier architecture reveals the similarities between XML document and view, DTD and BLU. These similarities will be exploited in the next subsection for detail. Here, the difference of the functions of DTD and BLU must be pointed out. DTD only specifies the constraint and structure that the data stream must comply, whereas BLU specifies server side behavior of the page object. In other words, BLU could be compiled into XML processor to associate actions with XML documents, while XML itself normally needs additional client-side scripts to associate actions.

XML processor is a software module that reads a XML document and provides access to its content and structure. XML processors typically process XML documents on behalf of applications. They normally integrate an available XML parser with DOM components inside. XML parser is available both in Java and C++. Document Object Model (DOM) provides programmatic access to the internal structure of XML documents. In other words, an XML processor is an engine installed in a web application to achieve the interoperability by exchanging universally formatted data streams in XML with other web applications.

Following the above introduction of XML, one can lead to a comparison of HTML with XML in Table 5.1. From that table, one can see that although there are differences in the way of formatting the data information, HTML and XML are still on the same level in the sense of that both of them serve as client end technologies. The only difference is that a HTML document will mostly be transmitted to a web browser and finally to a human end user, but a XML document will mostly be transmitted to a web application like an agent to automate the data exchanging and processing. That is why XML needs a DTD or
Table 5.1: Comparison of HTML and XML

<table>
<thead>
<tr>
<th>Language</th>
<th>Purpose</th>
<th>Audience</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>One set of markup tags</td>
<td>presentation</td>
</tr>
<tr>
<td>XML</td>
<td>self-defined markup language</td>
<td>syntax and semantics delivery</td>
</tr>
</tbody>
</table>

schema to ease the parsing by other web applications and also needs to deliver the logical meaning instead of visual presentation of data. In some sense, a human brain is also a data-processing machine. But it is a much more sophisticated machine that can understand visual information. As HTML is created to feed the information extraction by a human brain, XML is just created to feed logical and meaningful data to low-end processing units like computer web applications and to make them understand. Since in PCR, views are delivered for the clients no matter what contents and client technologies are encapsulated in it, XML should fit fairly well in this component category as will be shown in section 5.2.2.

In one word, XML is only one part of client technology to represent data stream, while PCR is a server side technology to construct service modules to feed clients with data presentation or data stream with syntax and semantical structures. There does exist a similarity between the schema of XML with the grammar of the data area of BLU of the PCR page object. This will be exploited by PCR in the next subsection to export XML document from the page object automatically.
5.2.2 Fitting XML in PCR Scenario through Messaging

While PCR provides a general conceptual model for the abstraction of server-side web applications, there is still an element missing in the whole client/server web infrastructure. That is the interfacing and messaging between web applications. Currently in PCR scenario, the interfacing among the pages within one application is fulfilled through the direct access of other page object's input data area, and the interfacing from the user to the application is realized mostly in CGI protocol mode. But in a more general sense of web applications, the web applications which may exchange messages with PCR constructed server not only include web browsers, but also preprogrammed intelligent agent and other PCR constructed large service modules. Furthermore, the amount of the data exchanged may be so large and the structure involved in the data stream may be so complex that the conventional interface methods just can not be competent for this job. In order to achieve convenience for large amount of data exchange while still keeping the interoperability between PCR constructed servers and web applications from other parties, XML comes in as an ideal candidate to provide the logical structure to the data streams that are exchanged between web applications. As a universal specification for the syntax and logical structure of on-the-fly documents, XML has the capability to naturally set up a on-the-site protocol between two communicating parties by exposing the logical structure of the data stream to both parties. The communicating applications can then use an existing XML parser and a DOM tool to understand and get access to the data in the stream.

In order to achieve the universal interoperability shown in Fig.5.2, one obvious way is for PCR to export the logical structure of its page object to XML document. After a close
examination of XML data modeling syntax and the grammar for the data area of BLU for each page object, the great similarity makes it very easy to translate the dynamic content of the page object to a XML document starting from the symbol table of the page. One step more, in the future expansion of PCR, PCR will provide one more optional view for each page: a logical view. A logical view is different from other normal views in that it is a pure XML document describing the logical structure of the data in the page object while others are presentational views that only display the data in some form without caring about the real syntax and semantics of the data. This logical view can be picked up by external web applications to understand the content of this page object. Just as presentational views are composed by HTML mixed with some other client technologies and PCR dynamic tags, the text file of logical views are composed by XML and PCR dynamic tags. This is exactly the situation shown in Fig.2.8 for the general view in Section 2.3.

In a reverse way(Fig.5.3), the future version of PCR will also expand the input capability
for each page object from only accepting data in CGI protocol to accepting XML based messages. PCR will enable the user with an additional option to integrate a XML parser and a DOM tool in the PCR generated service module so as to retrieve large amount of datum in XML document from other web applications.

Figure 5.3: Importing XML Message to Page Object
Chapter 6

Future Extension

The future development of PCR could be in many directions. One of the obvious work that needs to be done is a new input grammar or input data type to accept the input from a XML file and automatically deliver the reply file in XML[36] file using the data in BLU. Besides the emergent marriage of PCR with XML, two other important directions have also been envisioned. One is Hyper Component Library(HCL) and the other is Visual Development Environment on Web.

6.1 Hyper Component Library

The idea of Hyper Component Library(HCL) is an extension of component library[40]. This library is made up of component servers who take the request from the PCR compiler according to the PCR source code and reply with the component core, or add-on methods. This process is a dynamic process. For the component server, it normally generates components dynamically using the information of the request. In the sense of intelligent code
server, HCL provides a more dynamic way to generate user customary components than C++ templates. HCL can provide components in either intermediate code level like in C++ source code or in a higher PCR level. The other appealing part of HCL is that the HCL can be distributed over the internet. In this sense, the PCR code will take the role as HTML file and compiler will be the browser. In the PCR source code, some components can be denoted in its URL hyperlinking to its component server. The PCR compiler will parse this hyperlink to send a request to a remote component server, get the code, and assemble it into the compiled PCR program. This way, PCR is a web language even at the library and the compilation level. This mechanism is shown in Fig. 6.1.

HCL not only makes the core compiler and the PCR source code small, but also maxi-
mizes the degree of scalability of PCR program over to the web. If the protocol between a PCR hyper component link and a component server is specified delicately and made public, all the programmers on the internet can contribute to PCR language which might pave a way to boost the expansion of PCR. Fig.6.2 shows component library layers supported by PCR. It reveals the future web application infrastructure paved by PCR.

More interestingly, because of this very dynamic and distributed nature of component servers, these servers can be developed using PCR. Using a small trick, put the component code instead of hyper text in the views of each BLU, the web application developed will server components instead of hyper text to the compiler not the normal web user. As a byproduct, this reveals that PCR is a very flexible server generating a language which does
not restrict itself only to serving the normal web content. By defining the business of BLU to be component servering, PCR is an ideal candidate to develop HCL.

### 6.2 Visual Development Environment On Web

The extension of PCR to a visual development environment is obvious just as the case of Visual Studio in MS Windows[42],[43]. It will provide the programmer with a systematic way to manage the project and a visual way to develop the application. But the difference here is that this VDE will be put on the web and developed in PCR. And more importantly, with the concept of HCL, the VDE will share the HCL on different sites as in Fig.6.3. This picture makes everybody think of CORBA. But be careful, component requesting and
component installation are carried out at compiling time, not at real-time. This way, with the whole web being the supporting component reservoir, VDE tries to be THE standard of web application development environment.
Chapter 7

Conclusion

PCR helps set up infrastructure of large-scale web application through the cooperation of a group of page objects, which in turn are based on more elementary components. It provides web developers with a methodology and an efficient tool to specify and generate the high-performance server modules without having to implement the server from scratch and know all the network communications and database interfacing details. The development efficiency achieved by PCR will be the most valuable feature for the web developer community who are soaring with the speed of e-commerce.

PCR is an open-end project. Just as seen in Chapter 6, the future expansion of PCR is unavoidable. It will coevolve with the application of PCR into real practices.
Appendix A

PCR compiler implementation

A.1 PCR Grammar

As we have seen, PCR is a page specification language with its service code in C-style. Conventional compiler construction techniques[45] are used here with symbol table to reveal the program structure and its target code in high level language. Below is a recapitulation of the grammar that was given throughout the earlier parts of the paper, for different layer of application separately. This grammar is acceptable to the YACC parser-generator[44].

A.1.1 PCR View Grammar

The view of page object is a mixture of any kind of client technology with a set of strictly defined dynamic tags. There are three categories of the dynamic tags. One is the showing tags, each starts with ⟨!DTAG, followed by the object name to show and ended by ⟩. One is the control tags, such as ⟨!FOR⟩, ⟨!ENDFOR⟩ and ⟨!IF⟩, ⟨!ENDIF⟩ pairs. The other is
the statement tags, also started with \texttt{!DTAG}, followed by the statements and ended by \texttt{).}.
The statements are added to make it possible for the sophisticated interfaces. Inside each environment, the grammar is similar to the grammar used in service logic, which will be listed in detail in the next subsection.

\subsection*{A.1.2 PCR Service Logic Grammar}

\begin{verbatim}
pcr:
    START session page-list END
    START page-list END

session:
    session-head st-body }

session-head:
    CSESSION {

page-list:
    page-list page-rec

page-rec:
    page-head page-sb }

page-head:
    CPAGE id {

page-sb:
    session-obj page-body
\end{verbatim}
page-body

session-obj:

SESSION ;

page-body:

page-b

page-b mf-list

page-b:

struct-com

struct-com init-han

import page-var-list struct-com init-han

struct-com:

struct-com struct-types

struct-types

struct-types:

in-struct

struct

list

ref

variables

views

database

escape
views:
  view var-list ;

view:
  VIEW

ref:
  type var-list ;

type:
  REF id

list:
  list-head structure

list-head:
  LIST

import:
  IMPORT pname-list ;

pname-list:
  pname-list , id

id

page-var-list:
  page-var-list page-vars ;

page-var-com:
  page-var-start page-vars ;

page-var-start:
EXTERN id

page-vars:
  page-vars, id
  id

database:
  db-st var-list ;

db-st:
  DB

variables:
  var-st var-list ;

var-st:
  INT

var-list:
  var-list, id
  id

in-struct:
  in-head structure

in-head:
  IN_STRUCT

struct:
  struct-head structure

struct-head:
STRUCT

structure:

st-head st-body }

st-head:

id {

st-body:

st-body field

field

field:

int id ;

double id ;

char id [ num ] ;

init-han:

init handler-list

handler-list

init:

init-head mf-m statement-list } ;

init-head:

INIT {

mf-list:

mf-list mf

mf
\textit{mf:}

\textit{mf-head mf-arg-list mf-m statement-list }

\textit{mf-head mf-m statement-list }

\textit{mf-m:}

\}

\textit{mf-head:}

\textit{VOID id (}

\textit{int id (}

\textit{double id (}

\textit{char *id (}

\textit{id id (}

\textit{mf-arg-list:}

\textit{mf-arg-list comma arg}

\textit{arg}

\textit{comma:}

\textit{COMMA}

\textit{arg:}

\textit{int id}

\textit{double id}

\textit{char *id}

\textit{handler-list:}

\textit{handler-list hl}
hl

hl:

hl-head hl-body }

hl-head:

hl-start {

hl-start logical-expr ) {

hl-start:

HANDLER (

logical-expr:

logical-expr cat-opr condition

condition

cat-opr:

@@

con-srb:

)

con-srb:

)

condition:

con-slb logical-expr con-srb

cvar rel-opr cvar

cvar rel-opr num
cvar rel-opr string

rel-opr: one of

== > < >= <= !=

hl-body:

statement-list

statement:

if-statement
ifelse-statement
ifelseif-statement
for-statement
while-statement
do-statement
update-statement
goto-statement
new-statement
assign-statement
str-assign
void-function
return-statement
input-statement
escape

escape:
ESCAPE

input-statement:

id . GETINPUT ( ) ;

return-statement:

return expression

return:

RETURN

new-statement:

PAGE id assign NEW id ;

DB id assign NEW id ( string ) ;

str-assign:

cvar ASSIGN string ;

string:

string PLUS STRING

string PLUS cvar-string

STRING

cvar-string

cvar-string:

$ ( cvar )

assign-statement:

assign-head expression ;

assign-head fun-r ;
assign-head db-func;

void-function:

func-r;

func-r:

func-name);

func-name id-list);

func-name).id;

func-name id-list).id;

func-name:

id(

id.id(

id-list:

id-list,operand

operand

assign-head:

cvar ASSIGN

dexpression:

dexpression p-m mulexp

mulexp:

mulexp t-d primary

primary

primary:
ex-slb expression )

u-m primary

operant

ex-slb:

)

u-m:

-

p-m:

+

-

t-d: one of

* / operant:

num
cvar
string
cvar:

id

id . id

id . id

id . id . id

update-statement:

UPDATE id ;
UPDATE id . id ;

UPDATE THIS ;

UPDATE THIS . id ;

UPDATE THIS . id cvar ;

goto-statement:

SHOW id ;

GOTO id ;

SHOW id . id ;

REDIRECT cvar ;

db-func:

id . retrieve ( id , string )

id . retrieve ( id , cvar )

id . search ( id , string , cvar , cvar , cvar )

id . search ( id , string , cvar , cvar , cvar , cvar )

id . insert ( id , id )

id . update ( id , id , id)

id . delete ( id , id , id)

ifelseif-statement:

if-statement elseif-list else-statement

elseif-list:

elseif-list elseif-list else-statement

elseif-statement
elseif-statement:

    elseif-head statement-list }

elseif-head:

    elseif-start logical-expr ) {

elseif-start:

    ELSE IF ( 

ifelse-statement:

    ELSE IF ( 

else-statement:

    else-head statement-list )

else-head:

    ELSE ( 

if-statement:

    if-head statement-list )

if-head:

    if-start logical-expr ) {

if-start:

    IF ( 

for-statement:

    for-head statement-list }

for-head:

    expression-to expression ) {
expression-to:

  for-start expression TO

for-start:

  FOR ( id ASSIGN

while-statement:

    while-head statement-list }

while-head:

    while-start logical-expr ) {

while-start:

    WHILE ( 

do-statement:

    do-start statement-list do-mid do-tail

do-start:

    DO ( 

do-mid:

    { WHILE ( 

do-tail:

    logical-expr ( ;
A.2 Symbol Table

The symbol table of PCR is represented in a tree structure. All the symbols in a PCR program are PCR objects. They include basic type objects, methods, structure and PCR pre-defined or extended components like CPAGE, LIST and DB. Each of these objects is represented as a node in our symbol tree. The root of this symbol tree is the PCR program itself. The root can contain lots of objects, like CPAGEs and other components. Each component can recursively contain other components. If the functional relations reflected by the VIEW and GOTO are added to this tree, the symbol tree would be expanded to a graph. Current version of PCR only catch the relation revealed by VIEW and GOTO between CPAGEs.

Symbol table contains all the symbols, and parential and functional relations enforced by PCR. It is generated in the first pass by a PCR compiler. In the second pass, the PCR compiler will use the symbol table as an abstract representation of PCR program to construct the intermediate code and do the type checking, constraint and methodology enforcement.

The key members of the symbol node are:

```c
struct sym_node {
    char *name;
    char *clone_from;
    int component_type;
    int alloc_limit;
};
```
struct sym_node *child;
struct sym_node *next;
int (*define)(char *object, char *method, char *args);
int (*call)(char *object, char *method, char *args);
}

A.3 Intermediate Code Generation

The intermediate code currently supported by PCR is ANSI C++. The main task of the code generation is to transform the symbol tree to a C++ structured server module and translate and plug-in the statements at the right time. The timing is very important in this process. That is the reason why in the first pass of PCR program the compiler does not translate the statements. All the type-checking and translation of PCR statements are carried out in second pass after the whole skeleton of the server module has been obtained by transforming from the symbol tree. The statements or the service action codes are special leaves in the symbol tree. They will be filled in the second pass when the main skeleton is there.

More sophisticated one-pass compiler technique is avoided here since the two-pass way is more robust and conceptually clear and expandable. The current "symbol graph -> server module skeleton in C++ -> fill in the service action -> full-blown application server" compiling assembly makes it easy to compile the code into other high-level languages. All these transformations step from the same symbol graph which represents the essential topology of a PCR enforced application server program.
In another aspect, two-pass analysis makes the big picture of the whole program available to the compiler before it really translates the actions. This makes it feasible for PCR to support more problem domain functional relations. PCR could utilize the global information supplied by the generated symbol graph to do the optimization by restructuring the graph before the second pass takes place. Furthermore, PCR compiler could also do the constraint checking, such as checking the dead-loops existing in the program graph.
Bibliography


