Open Pluggable Edge Services
Internet-Draft
Expires: March 16, 2004

A. Beck
Lucent Technologies
A. Rousskov
The Measurement Factory

September 16, 2003

P: Message Processing Language
draft-ietf-opes-rules-p-00

Status of this Memo

Copyright Notice

Abstract

P is a simple configuration language designed for specification of message processing instructions at application proxies. P can be used to instruct an intermediary how to manipulate the application message being proxied. Such instructions are needed in an Open Pluggable Edge Services (OPES) context.

Table of Contents

1. Introduction

The Open Pluggable Edge Services (OPES) architecture [I-D.ietf-opes-architecture], enables cooperative application services (OPES services) between a data provider, a data consumer, and zero or more OPES processors. The application services under consideration analyze and possibly transform application-level messages exchanged between the data provider and the data consumer. OPES processors need to be told what services are to be applied to what application messages. P language can be used for this configuration task.

In other words, P language primary objective is to express statements similar to:

```
if message meets criteria C,
then apply service S;
```

Figure 1

Thus, P programs mostly deal with formulating message-dependent conditions and executing services.

P design attempts to satisfy several conflicting goals:

flexibility: Application intermediaries deal with a wide range of applications and protocols (SMTP, HTTP, RTSP, IM, etc.). The language must be able to accommodate virtually all known tasks in selecting a desired adaptation service for a message of a known application protocol (and conceivable future applications).

efficiency: Language interpretation must be efficient enough to be comparable with other message processing overheads at a typical application proxy (e.g., interpreting HTTP headers to determine response cachability).

simplicity: Typical configurations must be easy to write and understand for a typical OPES system administrator.

correctness: Many message handling configurations are written without direct access to intermediaries that will use those configurations. The extent of off-line (compile-time) correctness checks should catch all syntax errors and many common semantic errors such as undefined values and type conflicts.

compactness: It is possible that some processing instructions will be piggybacked as headers/metadata to messages they refer to, placing stringent size requirements on language code.

security: It should be difficult if not impossible to write malicious code that would result in security vulnerability of compliant language interpreter.

While P addresses OPES needs, its design is meant to be applicable for a variety of similar intermediary configuration tasks such as access control list (ACL) specification and message routing in proxy meshes or load-balancing environments.

P design is based on a minimal useful subset of features from several programming languages such as R (S), Smalltalk, and C++. Technically speaking, P is a single-assignment, lazy evaluation, strongly typed functional programming language.

2. Syntax

P syntax is defined by the following Augmented Backus-Naur Form (ABNF) [RFC2234]:

```
code = *(statement ";")

statement = assignment / function-call / if-statement

assignment = identifier ":=" expression

if-statement = "if" "(" expression ")" "{" code "}"

expression =
     name / function-call / "{" code "}"
     ... ; more to be defined (logical and arithmetic expressions)

name = identifier *( "." identifier)

function-call = name "(" [params] ")"

params = expression *( "," expression)

identifier = ALPHA *(ALPHA / DIGIT / "_")

... ; more primitives to be defined as needed
```

Figure 2

XXX: add /* comments */.

3. Language elements

3.1 Objects

P is centered around the concept of an "object" that is similar to objects from other object-oriented languages. An object is, essentially, a piece of data or information. The value of an object is indistinguishable from the object itself. Object type is defined by the semantics of applicable operations and manipulations. Almost everything in P is an object, even a piece of code. Here are a few P objects, listed one per line:

```
0
"http://www.ietf.org/"
Core
{ a := 1/0; }
```

Many objects contain other objects, often called members. Members are accessible by their name, using the member access operator (".). Member access operator has a single parameter: the name of the member to access. All P objects support "." operator, but not all objects have members. Here are a few examples:

```
Http.message.headers
Core.interpreter.stop
"string".nosuchmember
```

Many objects support operators other than member access. For example, member objects that support function call "()" operator are often call methods.

```
Http.message.headers.have(header)
Core.interpreter.stop()
1 / 0
"string" + "string"
```

P operators are described in Section 3.2. below.

P does not have built-in facilities for describing object types. When writing a P program, only objects known to interpreter (e.g., Core) and objects generated by known objects (e.g., Core.import("Http")) can be used. P supports loadable modules that can be used to add objects to support new application protocols. In fact, P core supports no application protocols directly. Instead, modules like "Http" can be used to process messages depending on application protocol being proxied.

No default (silent) object type conversion is supported. However,

explicit conversion (casting) is rarely needed because many methods are polymorphic (can work with several object types).

3.2 Operators

Several operators are used in P to denote common operations. These symbols are deemed to improve readability of P code as compared to their spelled-out-in-English counterparts.

P Operators

operator	default semantics
A == B	A is semantically equal to B; does not modify A or B.
A != B	semantical inequality, same as !(A == B).
!A	logical negation, same as (A == false)
A and B	logical concatenation, same as !(!A or !B)
A or B	logical disjunction (inclusive), same as !(!A or !B)
A + B	sum of A and B; does not modify A or B.
A * B	product of A and B; does not modify A or B.
A - B	difference between A and B; does not modify A or B.
A / B	ratio of A to B; does not modify A or B.
A.n	access to A's member named n; does not modify A; fails if A has no member named n.
A(...)	object A is to perform a function call with zero or more parameters; may modify A and/or parameters

Operator precedence defines natural evaluation order used in mathematics and many programming languages. In the following list, operators are ordered based on their precedence. Operators with smaller precedence index are evaluated first. Operators with the same precedence index are evaluated in the left-to-right order of occurrence in an expression.

1. .
2. ()
3. !
4. * /
5. + -
6. == !=
7. and
8. or

Except for the member access operator (".") operators do not have to be supported by an object. Moreover, operator semantics may differ from one object to another (or even from one invocation to another for the same object though the latter is unlikely to be common in practice). Object writers SHOULD follow common operator semantics and MUST document actual operator semantics when adding support for these operators to their objects. The interpreter MUST NOT allow object writers to change operator precedence.

Operators are not global special symbols but are passed to the object for interpretation, along with their parameters. Applying an operator is semantically equivalent to calling an object method. For example, the following three expressions are equivalent:

```
a + b + c
(a.+(b)) + c
(a.+(b)).+(c)
```

Figure 6

The "a + b + c" form is preferred for purely visual reasons. Core P module provides basic objects and operators for them (e.g., boolean and integer). Application-specific modules usually provide applications-specific objects; those objects usually have application-specific methods and may not have methods to support operations common for basic types. For example, an Http module supplies an HTTP header object that does not have a "*" method.

3.3 Expressions

P expressions are used in if-statements to specify the condition for the if-statement body to be interpreted.

```
if (Http.request.method == "GET" and time.current() > time.noon) {
        ...
}
```

Figure 7

Evaluation of an expression stops when the value of an expression is known and cannot be changed by further evaluation. This short-circuiting optimization technique is common to many programming languages. In the following example, the value of A will never be interpreted when C is interpreted, regardless of the context where C is used:

```
C := false and A;
if (C) { ... };
if (!C) { ... };
...
```

Figure 8

3.4 Statements

Objects are manipulated using if-statements and function-calls.

```
if (Http.request.method == "GET") {
        Services.applyOne(serviceFoo);
}
```

Figure 9

3.5 Assignments

Most procedural programming languages use variables to store intermediate processing results. In such languages, a variable is essentially a named piece of memory that can be assigned a value and can be updated with new values as needed. P does not have such variables. Instead, P uses a "single assignment" approach: an expression can be tagged with a name and that name can be reused many times in the program. On the surface, this is equivalent to having all "traditional" variables declared as "constant". The following two if-statements are semantically equivalent in P:

```
if (Http.request.headers.have(Http.makeHeader("Client-IP"))) {...}

h := Http.makeHeader("Client-IP");
hs := Http.request.headers();
```

```
if (hs.have(h)) {...}
```

Figure 10

If the expression changes, a new name must be used to tag the new expression. After an assignment statement, the value of the name is not the value of the expression, but the expression itself. Thus, the following two code fragments are equivalent and make no sense in P (the first fragment would make sense in languages such as C++):

```
h := Http.makeHeader("Client-IP");
h := Http.makeHeader("Server-IP");

h := Http.makeHeader("Client-IP");
Http.makeHeader("Client-IP") := Http.makeHeader("Server-IP");
```

Figure 11

The interpreter can but does not have to evaluate the expression named in the assignment statement until the name is actually used in an expression that requires evaluation (e.g., as a parameter of a function call statement). This allows for optional performance optimizations where only used expressions are evaluated.

P does not have user-defined functions. However, some code reuse is possible because P code is a valid expression and, hence, can be named and reused:

```
code := { ... complicated service action ... };
if (condition1) { code; };
...
if (condition2) { code; };
```

Figure 12

XXX: document whether expression has to be evaluated in the assignment context or use context. Document name scope.

4. Modules

Application-specific support is available in P via modules. Basic P primitives such as integer types and boolean operations comprise the Core module. Module is an object. The Core modules supplies the following methods to manipulate other modules:

Core.import("M"): load a module called "M" and return it as the result.

Core.lookup(M): start looking up unresolved attributes and method identifiers in a previously loaded module M.

The Core module is assumed to be loaded (and being looked up) before the interpretation starts.

XXX: document lookup conflict resolution.

5. OPES Services

Services module contains basic attributes and methods for searching and executing OPES services:

Services.findOne(URI): returns a service object that corresponds to the specified URI. Fails if no corresponding object exists.

Services.applyOne(service, ...): applies the specified service to the current application message and optionally supplies service-specific application parameters. XXX: should parameters include the part of the message to be modified or just services metadata?

Here is a service application example for a German to French translation service:

```
Http := import("Http");
if (Http.response.language_is("german")) {
        service := Services.find("opes://services/tran/german/french");
        service.toDialect("southern");
        Services.applyOne(service, Http.request.headers);
}
```

Figure 13

XXX: explain how failures are propagated and can be handled

XXX: add Core.interpreter.stop and Core.interpreter.restart methods.

6. Failures

Virtually any P statement may fail: expression denominator may be zero, named members may not exist, objects may not support applied operators, service execution may fail, interpreter may ran out of resources during an assignment, etc. A failure immediately stops interpretation of the first surrounding code block and assigns that block a boolean value of false.

If the failed block is a part of a larger expression, the interpreter MUST continue evaluating the expression containing the failed block using usual expression evaluating rules, including short-circuiting boolean expressions. If the failed block is a stand-alone statement, that statement fails and the failure is propagated using the above rules. If the implicit code block surrounding the program fails (XXX: document or require an implicit surrounding block like XML does), the entire P program interpretation terminates with a failure.

Failure propagation rules allow to catch failures, similar to an exception mechanisms in languages like C++ or Java, except that P exceptions are not objects (they carry no information). For example, here is a simple way to introduce a backup/failover service:

```
{
        ...
        Services.applyOne(unsafeService);
} or {
        ...
        Services.applyOne(failoverService);
};
```

Figure 14

The following example illustrates how a failure-prone service can be retried twice if needed:

```
code := {
        /* code executing the service */
};
code or code or code;
```

Figure 15

It is possible to force the interpreter to fail using the "Core.interpreter.fail(reason)" call. This is handy when there is a logical failure that the interpreter cannot detect on its own:

```
{
```

```
                        /* large piece of code executing several services,
                           each manipulating the current HTTP message ... */

                        /* checkpoint */
                        if (!Http.message.headers.have("Content-Length")) {
                                Core.interpreter.fail("services did not set
CL");
                        }

                        /* OK, continue message manipulation ... */
                } or {
                        /* recover from failure ... */
                }
```

Figure 16

7. Security Considerations

XXX: document non-obvious vulnerabilities: too many names, too deep nesting, invalid math, too much error logging; execution of unauthorized services, unauthorized exposure of sensitive information to authorized services.

8. Compliance

XXX: define what a compliant interpreter is.

Appendix A. Examples

This appendix contains half-baked examples to illustrate P usage in common OPES environments. Example themes are taken from [I-D.beck-opes-irml] to ease the comparison with IRML.

Here is a data provider example:

```
interpreter.languageVersion("1.0"); // fails if incompatible

Http := import("Http");
lookup(Http);

// Is the requested web document our home page?
isHome := request.uri.looksLikeHome();

// Does the user send us a specific cookie?
cookie := makeHeader("Cookie", "sew=23");
haveCookie := request.headers.have(cookie);

if (isHome and haveCookie) {
        Services := import("Services");
        service := Services.findOne("opes://local.net/add-lcl-
content");
        service.clientIp(request.clientIp);
        Services.applyOne(service);
}
```

Figure 17

Here is a data consumer example:

```
Services := import("Services");
service := Services.findOne("opes://privacy.net/priv-serv");
service.action("remove-referer");
Services.applyOne(service);
```

Figure 18

Appendix B. Change Log

Internal WG revision control IDs: $RCSfile: rules-lang.xml,v $ $Revision: 1.5 $.

Normative References

[RFC2234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[I-D.ietf-opes-architecture] Barbir, A., "An Architecture for Open Pluggable Edge Services (OPES)", draft-ietf-opes-architecture-04 (work in progress), December 2002.

Informative References

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[I-D.beck-opes-irml] Beck, A. and M. Hofmann, "IRML: A Rule Specification Language for Intermediary Services", draft-beck-opes-irml-03 (work in progress), June 2003.

Authors' Addresses

Andre Beck
Lucent Technologies
101 Crawfords Corner Rd.
Holmdel, NJ
US

Phone: +1 732 332-5983
EMail: abeck@bell-labs.com

Alex Rousskov
The Measurement Factory

EMail: rousskov@measurement-factory.com
URI: http://www.measurement-factory.com/

Intellectual Property Statement

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.