Matchmaker:
An Interface Specification Language for Distributed Processing

Michael B. Jones
Richard F. Rashid
Mary R. Thompson

19 December 1984

Abstract

Matchmaker, a language used to specify and automate the generation of interprocess communication interfaces, is presented. The process of and reasons for the evolution of Matchmaker are described. Performance and usage statistics are presented. Comparisons are made between Matchmaker and other related systems. Possible future directions are examined.

This paper also appears in the Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1985.

Technical Report CMU-CS-84-161

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Matchmaker:
An Interface Specification Language for Distributed Processing

Michael B. Jones
Richard F. Rashid
Mary R. Thompson

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Matchmaker, a language used to specify and automate the generation of interprocess communication interfaces, is presented. The process of and reasons for the evolution of Matchmaker are described. Performance and usage statistics are presented. Comparisons are made between Matchmaker and other related systems. Possible future directions are examined.

Keywords

Remote Procedure Call, Interprocess Communication, Object-Oriented Languages, Multi-Targeted Compiler, Interface Specification Language, Distributed Systems.

1. Introduction

One of the thorniest problems in building a distributed system is how to interface distributed system components. The earliest distributed systems required programs to directly manage a communication protocol on an I/O channel. Work on message-based systems often led to a style of interprocess interaction which stressed communication flexibility over interface correctness or ease of programming. Interfaces were effectively implemented in "message assembly language" as message records were explicitly packed and unpacked by user written code.

In recent years a number of distributed programming languages have been proposed. Design and implementation considerations have often been driven by abstract issues rather than the concrete requirements of a distributed system. Designing a single distributed programming language often ignores the fact that many applications are already written in existing languages such as C, Pascal and in the case of AI applications, LISP. Frequently such languages stress simple client/server communication and ignore the requirements of real-time system services.

Rather than being another distributed programming language, Matchmaker is an interface specification language for use with existing programming languages. It provides:

- a language for specifying object-oriented remote procedure call (RPC) interfaces between processes executing on the same machine or within the SPICE network,

- a multi-targeted compiler which converts these specifications into interface code for each of the major languages used within the SPICE environment, including C, PERQ Pascal [1], COMMON LISP [14] and Ada [4]. This code provides communication, runtime support for type-checking, synchronization and exception handling.

Matchmaker was started in 1981 as part of the CMU distributed personal computing project (SPICE [3]). It was built at first to automate some of the coding for the Accent[1]

[1] Accent is a trademark of Carnegie-Mellon University

operating system kernel [11] message interface, which forms the basis of the SPICE environment. It has evolved significantly during the last three years in its syntax, data representation semantics and communication semantics. At each point of change, decisions about the new Matchmaker design and implementation were driven by specific requirements of programmers in the SPICE environment.

Over the years, Matchmaker has proven to be a valuable tool. It has:

- eased implementation and improved the reliability of distributed programs by detaching the programmer from concerns about message data formats, operating system peculiarities and specific synchronization details,
- improved cooperation between system programmers working in different languages,
- enhanced system standardization by providing a uniform message level interface between processes,
- provided a language rich enough to express any data structure which can both be efficiently represented in messages, and reasonably represented in all target languages,
- reduced the cost of reprogramming interfaces in multiple languages whenever a program interface is changed.

Matchmaker provides a wide range of synchronization semantics ranging from synchronous remote procedure call to asynchronous message-style communication. A programmer can usually change the synchronization part of the Matchmaker specification without affecting the code which uses that interface.

Today, Matchmaker interfaces define all interprocess communication in the SPICE environment which consists of over 150 PERQ[2] computers communicating on an internetwork of several 3MHz and 10MHz Ethernets. Matchmaker is also used to specify and implement interprocess communication interfaces between PERQs and the CMU CS Department's 40 VAX[3] computers, which run a modified version of Berkeley 4.1bsd UNIX[4] [12] supporting Accent-style message communication. In all, Matchmaker has been used as the distributed programming support environment for over 500,000 lines of code written in four major languages. Matchmaker has evolved from a simple programming aid into the effective definition of interprocess communication within the SPICE environment.

In this paper we will discuss the Matchmaker language -- its syntax, data representation and communication semantics, and its implementation. We will also examine the issues which forced many of the important decisions in the Matchmaker design.

2. Language Overview

The computational model for Matchmaker consists of processes communicating with one another via messages. Messages are sent to communication ports. Accent ports, and rights to receive messages from specific ports, can be sent between processes in messages.

Ports also serve in a dual role as capabilities for objects. Matchmaker interfaces define operations upon those objects. Every remote procedure call specifies a destination port for the request. Thus, the ports may be viewed as tokens for instantiations of objects, and RPC requests to ports may be viewed as invocations of operations upon objects. Such an identification logically makes every Matchmaker request to a port an operation on the object represented by that port.

The syntax of Matchmaker specifications is fairly close to the Pascal or Ada specifications for the analogous objects. Constants of various types can be declared, new data types can be constructed from built-in types (within certain constraints), and remote procedures can be declared with a syntax fairly similar to Pascal procedures or functions. The invocation of a remote procedure on a port in a given target language usually consists of a procedure call, with that port as the first procedure parameter.

The built-in data types provided by Matchmaker are: **Boolean**, **Character**, **Signed** and **Unsigned Integers** of various

---

[2] PERQ is a trademark of Perq Systems Corporation

[3] VAX is a trademark of Digital Equipment Corporation

[4] UNIX is a trademark of AT&T Bell Laboratories

bit sizes, Integer SubRanges, Strings, Communication Ports, and Reals. New data types can also be constructed with some restrictions. Type constructor functions supported are: Records, fixed and variable-sized Arrays, Enumerations, Pointers to the above types, and certain kinds of Unions.

Representations for remote procedure arguments in messages are chosen by the Matchmaker compiler. Each message is assigned a unique id by Matchmaker, which is used at run-time to identify messages for a given interface. Once the message has been identified, the types of all fields within it are also known, since messages are strongly typed by Matchmaker at compile-time.

Certain semantic restrictions are placed upon the data types which can be declared to allow efficient passing of arguments in messages. In particular, pointers, variable-sized arrays, and unions can only occur in top-level remote procedure call declarations, and may not be used when constructing other types.

Several semantically different kinds of remote procedure call interactions can be specified in Matchmaker. The process normally initiating an operation is called the *client process*, and the process normally receiving requests is called the *server process*. The RPC paradigms provided are:

- **Remote_Procedure:** Generates code for a client process to send a request to a server, and to receive reply parameters back from the server. Timeout values can be specified, and the reply wait can be made asynchronous as well.

- **Message:** Generates code for a client process to send a single request message to a server without a reply.

- **Server_Message:** Generates code for a server process to send a single message to a client process.

- **Alternate_Reply:** Generates code for a server process to send a reply message back to a client process in response to a **Remote_Procedure** which is different than the normal reply message. **Alternate_Reply** messages are meant to be used for signaling exception conditions which occurred during execution.

Each of these varieties of calls except for Alternate_Reply takes a port to which to send the request as a parameter. Thus, "binding" is done dynamically on the basis of ports, and not by using some compile-time or link-time discipline.

3. Language Evolution

The Matchmaker program was originally conceived as a programming tool to simplify the sending and receiving of messages. It was planned to be a temporary expedient which would generate Pascal code until language intrinsics for sending and receiving messages could be added to the Pascal compiler. The intention to add interprocess communication support to our main programming languages was in line with contemporary distributed programming language proposals such as PLITS [5].

The original input to Matchmaker consisted of the names of the procedures to be generated and the list of parameters to each procedure. Associated with each parameter was an indication of the direction in which it was to be sent, the Pascal type of the parameter and the message-specific type description for the parameter. Since the only target language anticipated at that time was Pascal, the Pascal type declarations were imported into the generated code. This first version only generated simple synchronous remote procedure calls. Each generated procedure call would send a message to the server and then wait for a reply before returning to the client.

After several months of use it became apparent that the Matchmaker approach had several important advantages over the notion of adding language intrinsics to Pascal:

- The procedure-based form of Matchmaker generated calls made these interfaces easy to document and use.

- Such intrinsics would have to be added to each language which was to be used in the SPICE environment. Moreover, adding intrinsics would leave the project with the burden of supporting a non-standard version of each language.

- The input to Matchmaker could be developed into a language independent formal specification of the interfaces to the system servers.

This early version of Matchmaker also had its share of problems, however:

- The inflexible format of Matchmaker specifications made them difficult to use.

- Data type specification was awkward and too limited.

- The semantics of remote procedure call were too limited.

In response to these problems, the second version of Matchmaker allowed declaration of types in a Pascal-like syntax, the specification of some global message style options, and the specification of an arbitrary number of RPC interfaces. As Matchmaker was used to generate interfaces for more servers, variations on the remote procedure call were added. For example, the window management process was sent character strings to display on the screen. These messages did not require a reply message, so messages without replies were added. Some applications wanted to use remote function calls rather than procedure calls. Procedures that signaled their errors as exceptions were also added. Client and server processes were found to require completely asynchronous communication for some tasks. During this period, the evolution of Matchmaker was driven by specific demands made by the writers of server processes.

At this time, Matchmaker was widely used only by Pascal programmers and still required the inclusion of Pascal import files. The next major modification to the Matchmaker language came as a result of the need to use the Matchmaker specifications to generate C and LISP code. Since C is close to Pascal in style, it was possible to use Matchmaker to generate C code and to import the language types from C include files, instead of Pascal import files. The LISP implementors were not so fortunate. The usefulness of COMMON LISP was delayed by the need to hand code the LISP function to message interface for all the system servers. It was now obvious that a genuinely language independent specification was needed for the server interfaces.

The third and current version of the Matchmaker language fulfills this requirement. A Matchmaker specification now includes complete descriptions of the types of every argument that is passed. Matchmaker generates the target language (Pascal, C, LISP, etc.) type declarations to be imported into the generated code. The Matchmaker specification for a client/server interface is written in a formal language that is approximately as readable as Pascal type and procedure declarations. This specification is both the documentation of the interprocess interfaces, and the source code which is compiled into correct procedure calls and type declarations for the target language. The Matchmaker compiler is internally structured to allow the addition of code generators for other languages as they are added to the SPICE environment.

This same version of Matchmaker also includes enhancements which allow a fine grain of control over the message send/receive options. While older Matchmaker implementations made various assumptions about the manner in which messages were to be sent and received, it is now possible to control all such parameters, both statically and dynamically.

4. Usage and Performance

Given the extensive usage of Matchmaker within the SPICE system, there are a number of interesting statistics which are available on the use and performance of Matchmaker interfaces. The figures below were gathered from the PERQ Pascal interfaces to the standard SPICE server interfaces, including the Accent kernel.

Static Usage:

Number of Interfaces	15
Total Calls Declared	268
Total Asynchronous Requests	67
Total Alternate_Replys	7

Dynamic Usage:

No exact figures are available, but it is known that far more asynchronous (unacknowledged) calls take place than synchronous ones. This is due to the fact that most I/O activities such as screen, mouse, low-level keyboard, and Ethernet I/O are handled asynchronously.

Avg. Code Bytes Per Call:

	Client	Server
Min. (Accent Kernel)	146	165
Max. (Filesystem)	246	242
Avg.	212	181

It is significant to note that server interface code is usually smaller than the corresponding client code. This directly corresponds to the fact that more parameters tend to be passed into calls than are returned by them.

% Matchmaker Code by Size in Servers:

Min. (Authorization Server)	1.4%
Max. (Filesystem)	23%
Kernel	6.5%

The relative size of the interface code varies with the number of routines provided, the number of arguments passed, and the amount of processing requested by each call. The Kernel and Authorization Server each use one relatively simple interface, and respond directly to requests from clients. The filesystem, on the other hand, is implemented as a set of co-operating processes distributed across several machines, using several different interfaces. Hence, filesystem processes are clients of one another via inter-filesystem interfaces, as well as being servers, thus explaining the high percentage of interface code.

% Matchmaker Code by Size in Clients:

The percentage of total code in client processes which is Matchmaker code is not especially useful, since for all standard SPICE servers, client interfaces are imported from shared runtime libraries.

Avg. Message Passing Overhead with Arguments:

Bare Kernel Send & Receive	2.5 ms
Matchmaker Overhead	0.6 ms
Total Msg Passing Time	3.1 ms

Thus, Matchmaker interfaces account for ~ 19% of message passing time, when packing and unpacking the arguments into messages is included.

Message Passing Frequency:

Avg.	25 to 30 msgs / sec
Max. Ever Observed	110 msgs / sec

Msg Passing Time as % Total Time:

~ 30 msgs / sec
x 3.1 ms / msg
= ~ 9% total time spent in msg passing

Matchmaker Overhead as % Total Time:

~ 30 msgs / sec
x 0.6 ms Matchmaker overhead / msg
= ~ 1.8% total time in Matchmaker overhead

The currently generated Matchmaker code is known to be inefficient. Yet, it is important to note, as shown by the above statistics, that actual Matchmaker overheads in message passing have no perceivable effect upon system performance. If they were eliminated entirely, it would not be noticed.

Although all communication in SPICE is via messages and Matchmaker, we know of no normal system activity which is dominated by message passing time. Even at the maximum observed message passing rate, roughly 2/3 of the total time would still be used for other things. This is easily explained; it almost always takes longer to process the information passed in messages than the time it took to transmit it, given Accent's efficient message implementation.

Throughout several years of use, Matchmaker has permitted distributed applications to be built with nearly the same ease as single-process applications. Two examples illustrate this point:

- The SPICE window manager was originally written and debugged as a stand-alone application and then converted into a server process using Matchmaker. The conversion process required virtually no change to the underlying structure of the program.

- A set of autonomous file servers were converted into a distributed filesystem in less than a week by designing and implementing an appropriate Matchmaker interface between them.

5. Comparisons

Matchmaker can be most directly compared with Nelson's Diplomat [10] and the Cedar Lupine system [2]. Matchmaker, Diplomat and Lupine can all be described as remote procedure call stub generators. Both of these Xerox systems, however, were built around a single programming language. Lupine, in particular, uses the existing Mesa [9] interface modules as the basis for generating the remote procedure call stubs.

Matchmaker evolved during roughly the same period as Diplomat and Lupine. Matchmaker differs from these efforts in that it: is an external specification language, supports multiple languages in a heterogeneous machine and operating system environment, provides for a wide class of synchronization semantics in addition to remote procedure call, and supports an object-oriented computational model. Matchmaker is also unique in that it is the sole interface language for both local and network communication.

Matchmaker can also be compared with earlier attempts in the RIG [7] system to build generic interprocess interfaces. RIG provided a "Call" function which took as its arguments the object to be operated on (represented as a RIG process-port pair), the function to be invoked (message identifier), and the arguments. Matchmaker interfaces in contrast to the RIG approach are type checked, handle multiple languages in the style appropriate to that language and allow for greater flexibility in defining the information to be passed as part of a remote call.

Unlike the Argus's Actions and Guardians protocol [8], Matchmaker does not provide for atomic transactions. The nearest that a server can get to providing atomic transactions is to provide **Remote_Procedure** interfaces, with reply status values, and reply timeout values that cause blocking until a reply is received. These actions can then be known to be atomic in some cases. If the server cannot receive the message, the reply code is set to "Failure" and no action takes place. Likewise, if the server is reached, but can not successfully carry out the request, it will return a "Failure" code and abort the entire transaction. However, the hard case where a server is reached and then crashes before it completes the transaction, either leaves the client permanently blocked waiting to receive a reply, or returns a "Timeout" status, depending upon the options selected.

Unlike systems that are written entirely in one strongly typed language such as Argus/CLU and the Xerox systems, Matchmaker's type checking may be compromised by the language that invokes its interfaces. Matchmaker runtime code checks the types of the arguments that are extracted from messages but it must rely on its implemention language (Pascal, LISP, C, etc.) to guarantee the integrity of the values passed to it as parameters.

6. Future Directions

As with any evolving system, there is still substantial room for improvement in Matchmaker.

Matchmaker does not enforce a robust implementation of interprocess communication. Rather, it allows the implementer of a server process to choose from the underlying primitive communication paradigms provided, and to easily provide an RPC interface to the user.

A synchronous remote procedure call does not currently terminate when the target server process terminates abnormally. Instead, an exception or timeout is generated which must be handled by the client. It was originally felt that this was an adequate solution to the problem, but as more and more naive programmers use the system for developing their own applications, it has become apparent that handling such conditions can be difficult.

As a result, system work is underway to allow for a synchronous error return to be provided when a server crashes during the execution of a remote request. This support should be an appropriate mechanism for implementing truly atomic remote procedure calls. Likewise, enhancements were recently added which allow a fine grain of control over the message send/receive options. With these improvements a careful server implementer should be able to write a robust and transparent server interface that requires no particular sophistication on the part of the user of the interface.

The future direction of Matchmaker will probably be influenced by the development of SPICE applications that are implemented as closely co-operating servers distributed over more than one machine. Both the Sesame File System [6] and the TABS [13] Distributed Transactions manager are currently being implemented in this manner. The issues of robustness in the face of a remote server failure, and guaranteed response to the original client are being addressed by these servers.

7. Conclusions

Matchmaker is not a new distributed programming language; it is not a radical departure from existing techniques in the design and implementation of programming languages. Instead, Matchmaker is an important tool for distributed programming which has been evolving and in use for over three years. It has proven valuable and simple to use. It allows a server to automatically be accessible from clients written in any of the supported languages, regardless of the language in which the server is written. It permits distributed applications to be built with nearly the same ease as single-process applications.

Probably Matchmaker's greatest value is that it has become, in effect, *the* working definition of inter-domain communication in the SPICE system. Since it automates the implementation of RPC on top of messages, it is conceivable that different Matchmaker code generators could implement a similar form of RPC on a different communication medium, with almost no change to the client or server code involved.

Through use in real distributed systems, Matchmaker has succeeded in proving itself a useful tool for creating interprocess interfaces in a very demanding distributed environment.

I. Example Specification

The text which follows is a fictional Matchmaker interface specification for a "display server" process.

```
Interface Screen = 15000:        ! Base Msg ID is 15000

Constant
    Max_X       = 132;
    Max_Y       = 40;

    Inverted    = true;
    Normal      = not Inverted; ! A constant expression

Type
    Screen_Array = packed array [Max_X * Max_Y] of Character;

    Char_Vector  = * packed array [*] of Character;

    Screen_State = record
                       x         : byte;
                       y         : byte;
                       Reverse   : boolean;
                   end record;

    Screen       = port;   ! Port used for screen token

Message DisplayChars(
                         : Screen;
        x                : byte;
        y                : byte;
        chars [num]      : Char_Vector;  ! Note size parameter
        ) : No_Value;

Message PutChar( : Screen; c : Character) : No_Value;

Message ClearScreen( : Screen) : No_Value;

Remote_Procedure GetWholeScreen(
                         : Screen;
   out  ScreenArray      : Screen_Array;
   out  Current_X_Size   : byte;
   out  Current_Y_Size   : byte;
        ) : GR_Value;

Remote_Procedure SwapScreenState(
                         : Screen;
  inout State            : Screen_State;
        ) : No_Value;

Alternate_Reply No_Such_Screen;

End Interface
```

II. Example Matchmaker Output

This appendix contains the Matchmaker source and generated PERQ Pascal output for the client side of one call in an actual interface used in SPICE. The call presented is relatively simple in comparison to many of the calls used by the system. Essentially, it sends a communication port to a server, and receives a new port back from that server in a reply message.

The Matchmaker specification for the call is:

```
Remote_Procedure CopyEnvConnection(
                                : Port;
            OldConnection       : Port;
        out NewConnection       : Port_Ownership
    ) : GR_Value;
```

The generated Pascal code implementing the client interface depends upon the following data structures, which define Accent messages and type descriptors within messages. They are presented here without explanation (see [11] for details) in order to clarify the code to follow.

```
TypeType = packed record
             case integer of
               1: ( TypeName        : Bit8;
---                 TypeSizeInBits  : Bit8;
                    NumObjects      : Bit12;
                    InLine          : boolean;
                    LongForm        : boolean;
                    Deallocate      : boolean
                  );
               2: ( LongInteger : long)
           end;

Port     = long;

Msg      = record
             SimpleMsg  : boolean;
             MsgSize    : long;
             MsgType    : long;
             LocalPort  : Port;
             RemotePort : Port;
             ID         : long;
           end;
```

Finally, the generated client code for sending the request, and receiving the reply is as follows. ("`ReplyPort`" is an Own variable; "`{$IFC ... }`" is conditional compilation; "`WordSize`" is like "`sizeof`" in C.)

```
Function CopyEnvConnection(
            ServPort        : Port;
            OldConnection           : Port;
        var NewConnection           : port
    ): GeneralReturn;

type
    MyMessage = record
        head            : Msg;
        IPCNam2         : TypeType;
        Arg2            : port;
    end;

type
    RepMessage = record
        head            : Msg;
        RetCodeType     : TypeType;
        RetCode         : Integer;
        IPCNam3         : TypeType;
        Arg3            : port;
    end;

  var
    MyMsg       : MyMessage;
    RepMsg      : RepMessage;
begin

    with MyMsg.head do
    begin
        SimpleMsg := false;
        MsgSize := WordSize(MyMsg)*2;
        MsgType := NORMALMSG;
        RemotePort := ServPort;
        LocalPort := ReplyPort;
        ID :=     1604;
    end;
    with MyMsg do
    begin
{$IFC FastAssign then}
        IPCNam2.LongInteger := #2000220006;
        {$ELSEC}
        IPCNam2.InLine :=   TRUE;
        IPCNam2.Deallocate :=  FALSE;
        IPCNam2.LongForm :=  FALSE;
        IPCNam2.TypeName := TypePt;
        IPCNam2.TypeSizeInBits := 32;
        IPCNam2.NumObjects := 1;
        {$ENDC}
        Arg2 := (OldConnection);
    end;
    with RepMsg.head do
    begin
        MsgSize := WordSize(RepMsg)*2;
        LocalPort := ReplyPort;
    end;
    GR := Send(MyMsg.head,0,WAIT);
    if GR <> Success then
    begin
        CopyEnvConnection := GR;
        exit(CopyEnvConnection);
    end;

    GR := Receive(RepMsg.head,0,LOCALPT,RECEIVEIT);
    if GR <> Success then
    begin
        CopyEnvConnection := GR;
        exit(CopyEnvConnection);
    end;
    with RepMsg do
    begin
        if head.ID <>     1704 then
        begin
            CopyEnvConnection := BADREPLY;
            exit(CopyEnvConnection);
        end;
        {$IFC TypeCheck THEN}
        if RetCodeType.TypeName <> TYPEINT16 then
        begin
            CopyEnvConnection := BADREPLY;
            exit(CopyEnvConnection);
        end;
        {$ENDC}
        CopyEnvConnection := RetCode;
        {$IFC TypeCheck THEN}
        if IPCNam3.TypeName <> TypePtOwnership then
        begin
            CopyEnvConnection := BadReply;
            exit(CopyEnvConnection);
        end;
        {$ENDC}
        NewConnection := (Arg3);
    end;
end;
```

III. Matchmaker Language Syntax

The following is a syntax description of the MatchMaker language. Conventions used are as follows:

Double quotes (" ") denote literal tokens.
Square brackets ([ ]) denote optional productions.
Braces ({ }) are used to enclose a group of productions.
Three periods (. . .) denote optional repetition.
Vertical bars (|) separate choices between productions.
Parens (( )) are used to enclose comments.

Interface and Options Definitions

```
Specification
     ::= Interface_Spec
     ::= Types_Spec

Interface_Spec
     ::= Interface_Decl [Options_Decl]... [Data_Decl]...
         [Msg_Decl]... "End" "Interface"

Types_Spec
     ::= Types_Decl [Options_Decl]... [Data_Decl]...
         "End" "Types"

Interface_Decl
     ::= "Interface" Interface_Name "=" Msg_ID_Base ";"

Types_Decl
     ::= "Types" Interface_Name ";"

Interface_Name
     ::= Identifier

Msg_ID_Base
     ::= Integer_Constant

Options_Decl
     ::= "Options" {Option_Decl ";"}...

Option_Decl
     ::= Msg_Options
     ::= Protocol_Options
     ::= Ports_Options

Protocol_Options
     ::= "Protocol_Version" "=" Integer_Constant

Ports_Options
     ::= "Local_Ports" "=" {Integer_Constant | "*"}
     ::= "Ports_Backlog" "=" Integer_Constant
```

DataType Definitions

```
Data_Decl
     ::= Use_Decl
     ::= Type_Decl
     ::= Constant_Decl

Use_Decl
     ::= "Use" Single_Use...

Single_Use
     ::= Interface_Name "From" File_Name ";"

File_Name
     ::= String_Constant

Constant_Decl
     ::= "Constant" Single_Constant...

Single_Constant
     ::= Constant_Name "=" Constant_Expr ";"

Constant_Name
     ::= Identifier

Type_Decl
     ::= "Type" Single_Type...

Single_Type
     ::= Type_Name "=" Type_Specification ["," Type_Option]...
     ";"

Type_Name
     ::= Identifier

Type_Option
     ::= "TypeType" "=" Integer_Constant
     ::= "Deallocate" ["=" Boolean_Constant]
     ::= "NoDeallocate"
     ::= "Element_Size" "=" Integer_Constant
     ::= "Element_Count" "=" Integer_Constant

Type_Specification
     ::= Type_Name
     ::= Builtin_Type
     ::= Array_Type
     ::= Record_Type
     ::= Pointer_Type
     ::= Enumeration_Type
     ::= Union_Type

Builtin_Type
     ::= "Boolean"
     ::= "Character"
     ::= "Real"
     ::= Integer_Type
     ::= String_Type
     ::= Port_Type

Integer_Type
     ::= "Unsigned" ["[" Integer_Constant "]"]
     ::= "Signed" ["[" Integer_Constant "]"]
     ::= Subrange_Type
     ::= "Long"
     ::= "Short"
     ::= "Byte"

Subrange_Type
     ::= Integer_Constant ".." Integer_Constant

Port_Type
     ::= "Port"
     ::= "Port_Send"
     ::= "Port_Receive"
     ::= "Port_Ownership"
     ::= "Port_All"

String_Type
     ::= "Perq_String" ["[" Integer_Constant "]"]

Array_Type
     ::= [Packing] "Array" "[" Array_Size "]" "Of"
     Type_Specification

Array_Size
     ::= Integer_Constant
     ::= "*"

Packing
     ::= "Packed"
     ::= "Unpacked"

Record_Type
     ::= [Packing] "Record" Record_Component... "End" "Record"

Record_Component
     ::= Field_Identifier ":" Type_Specification ";"

Field_Identifier
     ::= Identifier

Pointer_Type
     ::= "^" Type_Specification

Enumeration_Type
```

```
    ::= "(" Enum_List ")"

Enum_List
    ::= Enum_Element ["," Enum_Element]...

Enum_Element
    ::= Enum_Name ["=" Integer_Constant]

Enum_Name
    ::= Identifier

Union_Type
    ::= "Union" "<" Union_Selector_Type ">" "Of"
        Union_Component... "End" "Union"

Union_Selector_Type
    ::= Type_Specification

Union_Component
    ::= Union_Tag ":" "(" [Record_Component] ")" ";"

Union_Tag
    ::= Constant_Expr
    ::= "Otherwise"
```

Message Definitions

```
Msg_Decl
    ::= Msg_Code_Decl
    ::= Msg_ID_Decl

Msg_Code_Decl
    ::= Msg_Body ["," Msg_Options]... ";"

Msg_Options
    ::= Msg_Param_Key "=" Integer_Constant

Msg_Body
    ::= "Message" Arg_List ":" Msg_Result
    ::= "Remote_Procedure" Arg_List ":" Msg_Result
    ::= "Server_Message" Arg_List
    ::= "Alternate_Reply" [Arg_List]

Msg_Result
    ::= Special_Result
    ::= Arg_Type

Special_Result
    ::= "GR_Value"
    ::= "No_Value"

Arg_List
    ::= "(" Msg_Arg [";" Msg_Arg]... ")"

Msg_Arg
    ::= Data_Arg
    ::= Special_Arg

Data_Arg
    ::= [Arg_Direction] Data_Arg_Spec;

Arg_Direction
    ::= "In"
    ::= "Out"
    ::= "InOut"

Data_Arg_Spec
    ::= Simple_Arg_Spec
    ::= Variable_Arg_Spec
    ::= Union_Arg_Spec

Simple_Arg_Spec
    ::= Arg_Name ":" Arg_Type

Variable_Arg_Spec
    ::= Arg_Name "[" Arg_Cnt_Name "]" ":" Arg_Type
    ::= "[" Arg_Cnt_Name "]" Arg_Name ":" Arg_Type

Union_Arg_Spec
    ::= Arg_Name "<" Selector_Name ">" ":" Arg_Type
    ::= "<" Selector_Name ">" Arg_Name ":" Arg_Type

Special_Arg
    ::= Special_Usage Arg_Name ":" Arg_Type
    ::= ":" Arg_Type

Special_Usage
    ::= Port_Usage_Key
    ::= Msg_Param_Key

Port_Usage_Key
    ::= "RemotePort"
    ::= "LocalPort"

Msg_Param_Key
    ::= "MsgType"
    ::= "ReplyType"
    ::= "Send_Option"
    ::= "Send_Timeout"
    ::= "Receive_Timeout"

Arg_Cnt_Name
    ::= Arg_Name

Selector_Name
    ::= Arg_Name

Arg_Name
    ::= Identifier

Arg_Type
    ::= Type_Name ["." Type_Option]...

Msg_ID_Decl
    ::= "Skip_ID" ";"
    ::= "Next_ID" "=" Integer_Constant ";"
```

Expression Syntax

```
Constant_Expr
    ::= Or_CTCE (Valid types context dependent)

Integer_Constant
    ::= Adding_CTCE (Must be integer valued)

Boolean_Constant
    ::= Or_CTCE (Must be boolean valued)

Character_Constant
    ::= Primary_CTCE (Must be character valued)

String_Constant
    ::= Primary_CTCE (Must be string valued)

Enumeration_Constant
    ::= Primary_CTCE (Must result in a declared Enum_Name
    identifier)

Or_CTCE
    ::= And_CTCE ["Or" And_CTCE]...

And_CTCE
    ::= Not_CTCE ["And" Not_CTCE]...

Not_CTCE
    ::= ["Not"] Relational_CTCE

Relational_CTCE
    ::= Equality_CTCE
        [ {">" | ">=" | "<=" | "<"} Equality_CTCE ]...

Equality_CTCE
    ::= Adding_CTCE [ {"=" | "<>"} Adding_CTCE]...

Adding_CTCE
    ::= [{"+" | "-"}] Multiplying_CTCE
        [{"+" | "-"} Multiplying_CTCE]...

Multiplying_CTCE
    ::= Primary_CTCE [ {"*" | "/" | "Mod"} Primary_CTCE]...

Primary_CTCE
    ::= Identifier
    ::= Constant_Lexeme
    ::= "(" Or_CTCE ")"
```

Lexical Definitions

```
Constant_Lexeme
     ::= Octal_Literal
     ::= Decimal_Literal
     ::= String_Literal
     ::= Character_Literal
     ::= Boolean_Literal

Octal_Literal
     ::= "#" followed by a non-empty octal digit string.

Decimal_Literal
     ::= A non-empty decimal digit string.

String_Literal
     ::= A character string enclosed in double quotes.  A
     double quote in a string must be doubled.

Character_Literal
     ::= A character enclosed in single quotes.  A single quote
     in a character literal must be doubled.

Boolean_Literal
     ::= "True"
     ::= "False"

Identifier
     ::= A string composed of letters, digits and the
     underscore character, not starting with a digit.
     Identifiers are matched in a non-case-sensitive manner.

Comment
     ::= At any lexical break, comments can be inserted as:
         "!" Arbitrary comment text <End_Of_Line>
```


Acknowledgments

The authors would like to express their thanks to the following people who helped in the design, evolution, and implementation of Matchmaker: Jeff Eppinger, Joe Ginder, Jim Large, Rob MacLachlan, Doug Philips, Keith Wright, and Mike Young. Thanks also go to Bob Fitzgerald, who provided some of the statistics for this article.

This research was sponsored by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

References

1. Miles Bartel, Michael Kristofic. PERQ Pascal Extensions. In *PERQ Software Reference Manual*, Three Rivers Computer Corporation, 1982.

2. Birrell, A. D. and Nelson, B. J. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39-59.

3. . Proposal for a joint effort in personal scientific computing. Tech. Rept. , Computer Science Department, Carnegie-Mellon University, August, 1979.

4. *Reference Manual for the Ada Programming Language.* July 1982 edition. Dept. of Defense, Ada Joint Program Office, Washington, DC, 1982.

5. Jerome A. Feldman. "High Level Programming for Distributed Computing." *Comm. of the ACM 22*, 6 (June 1979), 353-368.

6. Michael B. Jones, Richard F. Rashid, Mary Thompson. Sesame: The Spice File System. Carnegie-Mellon University, October, 1982. Internal Document

7. Keith A. Lantz, Klaus D. Gradischnig, Jerome A. Feldman, Richard F. Rashid. "Rochester's Intelligent Gateway." *Computer* (October 1982), 54-68.

8. Liskov, B. and Scheifler, R. Guardians and actions: Linguistic support for robust, distributed programs. Proceedings Ninth ACM SIGACT-SIGOPS Symposium on Principles of Programming Languages, ACM, January, 1982, pp. 7-19.

9. J.G. Mitchell, W. Maybury, R. Sweet. Mesa Language Manual. Xerox Research Report CSL-79-3, Xerox Research Center, Palo Alto, CA, 1979.

10. Bruce Jay Nelson. *Remote Procedure Call.* Ph.D. Th., Carnegie-Mellon University, May 1981.

11. Rashid, R. F. and Robertson, G. Accent: A Communication Oriented Network Operating System Kernel. Proceedings of the 8th Symposium on Operating Systems Principles, December, 1981, pp. 64-75.

12. D. Ritchie. "The Unix Time-Sharing System." *CACM 17*, 7 (July 1974), 365-375.

13. Alfred Z. Spector, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Abdelsalam Heddaya, Peter M. Schwarz. Support for Distributed Transactions in the TABS Prototype. Proceedings of the 4th Symposium on Reliability In Distributed Software and Database Systems, October, 1984. Also available as Carnegie-Mellon Report CMU-CS-84-132, July 1984.

14. Guy L. Steele Jr.. *COMMON LISP: The Language.* Digital Press, 1984.