An APL-based Relational Data Management Language Using SYSTEM R as Data Server

F. Antonacci, P. Dell'Orco

IBM Italy, Rome Scientific Center

Abstract

This report describes a system for accessing System R, via AQL, a query language based on APL, developed by IBM Italy Scientific Centers. System R and AQL reside in two different Virtual Machines: the DB Machine and the User Machine, respectively. The two Machines communicate through VMCF, an inter-machine communication facility in VM/370 environment.
Queries formulated in AQL are shipped to System R, translated into SQL (the native query language of System R) and the results brought back to the User Machine.
The major advantage of this architecture is easiness of implementation and separability between the two environments. From the user's standpoint, this approach combines a friendly and flexible query language with a powerful data manager.

1. Introduction

The relational model of data [4] has been widely accepted in the Data Base research community, mainly because of its conceptual simplicity, symmetry and data independence. Within this framework, a number of DBMS's [1, 3, 5, 6, 7] has been developed.

Each of them has stressed some particular aspect either of the management of the data or the user interface: some systems (e. g. System R) give major emphasis to the general performance of the system (efficiency, security, reliability, etc...); others (e. g. Query-by-Example) give a user a friendly interface or, like AQL, offer a homogeneous environment encompassing programming language and access to the data.

Therefore the choice of one of these systems means often renouncement of some characteristics in favour of others.
Our goal was to give to the user a system which is a reasonable compromise among the characteristics of System R and those of AQL. A system that combines a simple and flexible interface with an efficient and reliable data manager can be implemented by translation of the interface language into the language of the data manager.

AQL [1] is a query and data manipulation language. It can be considered a supersystem of APL [2], with user friendly characteristics, such as: use of default options, automatic navigation among tables, homogeneous environment with the host language (APL).

System R [3] is a data base management system (DBMS) with characteristics mostly intended for the efficiency, reliability and security of the data. Its query language is called SQL.
We propose therefore a system where System R is the DBMS and AQL the query and manipulation language.

The core of the system is a translator which transforms each AQL transaction into a corresponding set of SQL transactions, using APL as a programming language and monitor/interpreter for the communication between System R and AQL. This monitor allows it to send transactions from AQL to System R, to execute them and receive the results in the AQL environment. This approach will be illustrated by a series of examples of AQL transactions, translated into SQL.

The most important characteristics of System R we want to take advantage of are the following:

- efficiency in the execution of the transactions, due to the optimization of the access paths to data;
- possibility of multi-user updating, due to the choice of the minimum entity to keep in the update;
- security, due to the possibility to grant and revoke authorizations to specific operations on the data base;
- reliability, due to the checkpoint/restart procedures.

Together with these characteristics, the proposed system adds the user friendliness of AQL, which can be summarized as follows:

- use of default options, which allows the user to give the minimum information in writing transactions;
- use of menus, by means of which, the user is asked to give the missing information in case of ambiguity not resolvable by the system;
- use of synonyms, both of attributes or of keywords of the language, defined by the user;
- automatic navigation through relations, according to pre-specified paths during the definition of the relational schemes;
- conciseness in formulating transactions, by the possibility of using not only single data, but also arrays of data inside the transaction body, according to the APL philosophy;
- easy extensibility of the language, by naming transactions and adding new keywords;
- homogeneity with the host language, which allows both the use of APL elementary or user-defined functions in the body of a transaction and the use of transactions in the body of a user-defined function as well as the processing of results in an interactive way.

The approach is to implement a translator which transforms the AQL transaction in (a set) of SQL transactions, and to restructure their results according to AQL syntax and data structure.
In general, an AQL transaction corresponds to the iteration of one or more SQL transactions. In fact, despite the surface syntactic similarity between the two languages, the greater semantic power of AQL and its capability (inherited from APL) of treating in a "transparent" way arrays of data, requests both a translation and an execution control.

The cost of this choice is mainly due to the overhead for the translation; nevertheless, the portion of this cost due to the syntactic analysis of the translation is already present in the AQL transaction for its completion (defaults fulfilling, synonymy resolution, etc..).
The residual costs are then only those due to the execution controller and to the reshaping of the results: but these costs seem to be justified with respect to the union of the advantages of both systems.

In the following section, the architecture of the proposed system is exposed; after that, some cases of the translator action will be illustrated through examples.

2. Architecture

The system allows communication between two environments (AQL and System R or, more generally, APL and PL1) of which one of "interpreter" (AQL/APL) and the other of "compiler" type (System R/PL1); both working in an environment based on the concept of "virtual machine" under the control of VM/370 [8] operating system.

This problem may be approached in two ways: by means of an auxiliary processor that, from inside the same virtual machine, allows the passage of shared variables from one environment to another, or keeping the two environments in two different virtual machines and using an auxiliary processor to manage the communication between the two machines.

The first way offers advantages from the performance point of view (data are transferred in the same machine, through the memory), but it requires a virtual machine as large as the sum of the two machines, that tends to be penalized; the second way performs data transfer via supervisor, (with VMCF communication technique) with a loss of efficiency that is not perceived by the user, and requires smaller virtual machines, that are not penalized.

This fact, together with the particular operational environment, suggests choice of the second way. In any case the following considerations do not depend on the peculiar communication mode between environments and may be applied to both contexts.

To describe the functioning of this system, we can examine the processing of a transaction that is input from the machine containing the AQL system (call it AQL machine).

This transaction undergoes the syntactic analysis that extracts from it the corresponding parse tree, by controlling at the same time the syntactic correctness of the statement.

The parse tree is then traversed to fulfill the relevant default options, possibly interacting with the user through menus to resolve synonyms, to perform the navigation among tables, etc..

The result of this process is the completed and correct version of the transaction (canonical form) in the form of a parse tree; this is then processed by the proper translation process.

This process is in turn performed by traversing the canonical form and producing the SQL transaction scheme through the substitution of AQL elementary syntactic structures with the corresponding SQL structures. This process will be detailed later on.

The result of a translation is sent, through the auxiliary processor for communication (in our case, AP401 [9]) to the machine where System R resides (call it S/R machine).

Here it is received by a monitor which provides for creation and management of the PL1 structures that are to receive both the constant values present in the transaction and its result.

This monitor also activates and controls execution of the transaction whose result is structured in a few PL1 variables (one for each data type) and sent to the AQL machine.
In this machine it is subdivided in as many APL variables as the requested attributes are: this result has the same structure as the results coming from AQL transactions.

3. The AQL-SQL translator

The translator, as seen before, accepts the canonical form of the AQL transaction and builds the corresponding SQL transaction scheme.
The scheme of a given SQL transaction is obtained from the transaction itself by substituting the values (or constants) that appear in it, with the character '?' (question mark).

In the case that data in the transaction are elementary data (not arrays), the translator could deliver the transaction already containing data; but in general AQL transactions may not only contain single data, but also arrays of them.
In these cases, the transaction is conceptually equivalent to its iteration on each element of the array or of the cartesian product of the arrays of data present in the transaction.
To clarify the matter, let us consider the following simple AQL request

(*attr* OF *rel*) WHEN *attr1 fp val* ,

where *attr* and *attr1* are attributes of the relation *rel* , *fp* is any "comparison function" AQL (EQ, GT, etc..), OF and WHEN are AQL keywords and *val* contains a constant or an array of constants.

If *val* is a single value, the result of this request is constituted (conceptually) by a single list of values of *attr* that satisfy the condition; if *val* is an array of values, the result is constituted by a set of lists each of which corresponding to a different element of *val* , so simulating the iterative application of the request scheme to each element of *val*.
Note that in AQL (as in APL) the concept of "list" is represented by the concept of "array" (orthogonal set of data).

The SQL equivalent transaction (in the case *val* is a single value) is:

SELECT *attr* FROM *rel* WHERE *attr1 comp val* ,

where *comp* is the SQL keyword corresponding to *fp* , SELECT, FROM, WHERE, are SQL keywords and the other symbols preserve the meaning seen before.

An alternative way to put the same query is:

SELECT *attr* FROM *rel* WHERE *attr1 comp* ?

by applying this scheme to *val*.

If, on the contrary, *val* is an array of values, only the second alternative can be applied, as the scheme has to be applied iteratively to each element of *val*.
Further, at each iteration step, a subset of *rel* is delivered: these results have then to be shaped to conform to the AQL results structure.

The problem is more complicated if more conditions, connected by logical functions (AND, OR) are present.
To illustrate this case, consider this AQL request:

(*attr* OF *rel*) WHEN (*attr1 fp val*) AND *attr2 fp1 val1* ,

where *fp1* is a comparison function and *attr2* is an attribute of *rel*.
The structure of the result in this case depends on the dimensions of *val* and *val1*: if both are constituted by one value, a list of values of *attr* is obtained; if, on the contrary, at least one contains more than one element, as many sets of lists are obtained as the elements of *val* are, each one containing as many lists as the elements of *val1* are; each

one of these lists will be constituted by the values of *attr* satisfying both the conditions applied to each element of the cartesian product of *val* and *val1*.

For example, if *val* contains three elements and *val1* contains two elements, the result will be a three-dimensional array with three planes, two rows, and the suitable number of columns.
In the first plane, the first row will contain (identifiers of) the elements of *attr* satisfying the two conditions applied to the first element of *val* (i.e. *val(1)*) and of *val1* (i.e. *val1(1)*); the second (identifiers of) the values of *attr* for *val(1)* and *val1(2)* ; in the second plane, the first row will refer to *val(2)* and *val1(1)* , the second to *val(2)* and *val1(2)* , etc..
This process applies uniformly also when more logical functions are present.

In the case of SQL, to deal with these cases, a transaction scheme has to be necessarily produced and then iteratively applied to the cartesian product of the sets of values present in the conditions.
The SQL transaction equivalent to the one before seen is composed of two parts: the first, constituted by the scheme:

SELECT *attr* FROM *rel* WHERE *attr1 comp* ? AND *attr2 comp1* ?

(*comp1* is the SQL keyword for *fp1)*, and the second, called "combination table" constituted by the cartesian product of the elements of *val* and *val1*.

Both these results of the translation process are sent to the S/R machine, where monitor takes care of their execution, in the fashion already seen, and sends the results back to the APL machine.

In what follows operations on the data base is illustrated, along with their translation process.

3.1. Definition

The definition procedure in AQL has the following syntax:

dbname DEFINE *relname* ,

where *relname* is the name of the relation we want to create in the collection of relations called *dbname*.
The system asks interactively to the user the name of the attributes, as well as their characteristics, so filling a table which lists names and characteristics of the attributes of the relation we want to create.

In System R, collections of relations are held in spaces called "segments" that have to be acquired before data definition: the name *dbname* used in the AQL transaction may hence be given to this collection.

The translation of this class of transactions gives then the class of SQL transactions (in the following, a couple of square brackets is used to enclose optional strings):

CREATE TABLE *relname* (*attr1* ([*characteristics_of_attr1*]),
(*attr2* ([*characteristics_of_attr2*]),
.....)
IN SEGMENT *dbname*.

The list of attribute names and their characteristics are deduced from the table already seen.

Note that, as it will be better seen further on, there are substantial differences in behavior between S/R and AQL as far as the results of grouping operation are concerned: in fact S/R returns as result a relation that, as such, has no tuples repetition, while AQL gives as result a set of APL variables that may contain possible repetitions.
It may happen that these repetitions are wanted: this may be achieved making the translator to add, in the list of "grouping" attributes, a candidate key, in a way transparent to the user.
This key is, in any case, created by the system, during the data definition process, under form of an attribute with a system-reserved name.

To do that, this attribute is added to the attribute list of the preceding transaction. For simplicity, its name will be denoted by tids.
This attribute contains progressive integers, so identifying tuples.
In exactly the same way operations of update and deletion of the relational scheme are translated, as well as operations of creation and deletion of indices on attributes.

3.2. Data entry

In AQL, the operation of adding data to the relations can be performed in four ways:

- interactively by means of a terminal;
- by means of APL variables, containing the data to add;
- by means of query results;
- using bulk input files.

In the first case, the syntax of the AQL transaction is:

((*attr_list*) OF *rel*) ADD '',

where *attr_list* is the list of the attributes to which new data are to be added and *rel* is the relation name which they belong to.
The system answers by asking the new value, for each attribute.
Such values are collected in APL variables (one for each attribute in *attr_list*).
The translation of the transaction gives the following schema:

INSERT INTO *rel* (*listattr*): < *qmarks* >,

where *listattr* is the list of the attributes contained in *attr_list*, separated by commas, and *qmarks* is a list of question marks (one for each attribute), separated by commas.

Such a scheme is applied repeatedly by the executor (monitor) to the set of variables which contain the input data, replacing in an orderly way the question marks by the single values contained in the variables.

The second type of adding in AQL is, as follows:

((*attr_list*) OF *rel*) ADD *varname_list* ,

where *varname_list* is a list of variable names containing the data to be added.
The translation follows the criteria already seen; *varname_list* is used to retrieve the variables which replace the question marks, according to conditions before seen.

The third type of data adding has the following AQL syntax:

((*attr_list*) OF *rel*) ADD *query* ,

where *query* is an AQL request.

Such a type of transaction is translated into:

INSERT INTO *rel* (*listattr*) *SQL_query* ,

where *SQL_query* is a SQL request, whose treatment will be considered in the following.

The last case provides the massive adding of the data from secondary storage and is formulated as follows:

((*attr_list*)OF *rel*) ADD *fname* [*ftype fmode*],

where *fname*, *ftype* and *fmode* represent the name, type and mode of the bulk input file.

In the last case, the SQL data storage procedure BULKLOAD is called by means of the auxiliary processor that allows the communication between APL and CMS (AP100); this procedure is executed from the same AQL machine.

3.3. Queries

Unlike the data base operations seen until now, the queries can be formulated by means a very rich syntax, since they can link an undefined number of conditions by means of logical functions, can nest an undefined number of queries, etc...

Moreover, selection, projection and join operations, grouping and arithmetical operations can be performed, (counting, sum, etc...) inside the queries.
Formally, (as it will appear more clearly in the following) both query classes (with and without grouping) differ in the name of the last function that appears in their canonical form (except for the arithmetical functions).

We will deal separately with both queries groups, showing also the action of arithmetical functions.

3.3.1. Simple queries

The common syntax of an AQL query is:

< funct_ name > (*target_ list*) WHEN *condition_ list* ,

where *funct_ name* (optional) specifies the name of an APL elementary or defined function to be applied to the result of the query; *target_ list* is a list of this type:

(*attr1* WITH *attr2* WITH ....) OF *rel.*

This allows it to have a request list constituted of more than one attribute name catenated by the keyword WITH; *condition_ list* is a list as the following:

(*attr' fp val*) *fl attr" fp val1*..... ,

where *attr', attr",* are attributes belonging to *rel* or to other relations, *fp* is a "comparison function", *fl* is a logical function (AND, OR (the possible NOT precedes each single condition)), and at last *val, val1,* etc.. can represent a constant, a constant list, an APL defined variable, or a query.
If *rel* is not specified, the qualification of the attribute list is made by the system; if *condition_ list* is not specified, all the items of the requested attributes are retrieved; the attribute list can consist of one attribute only.

In the following we consider the translation cases of the main AQL syntactic constructs.

i) Queries with one condition

To show this case, we consider the following simple query:

(attr OF *rel)* WHEN *attr1 fp val* ,

where, for simplicity, we suppose that *attr1* belongs to *rel.*
The comparison function *fp* can be the function of "set membership" (in AQL: ISONEOF) or one of the functions: LT (less than), LE (less or equal than), EQ (equal), GE (greater or equal than), GT (greater than), BETWEEN (included in).

Supposing that *val* is a constant (or a list), the corresponding canonical form of the query is:

A1 ← *attr1* OF *rel*
A1 ← A1 *fp val*
A2 ← *attr* OF *rel*
A1 ← A2 WHEN A1

If *val* represents a query, the canonical form contains in the first part the analysis of the query corresponding to *val* and in the second part the canonical form already seen, where *val* is substituted by the name of the temporary variable, to which the result of the first part is assigned.
Beginning from the last statement, the translator substitutes the AQL keywords with corresponding SQL ones and uses the temporary variables to select the next statement to translate.

If *fp* is a membership function (ISONEOF), and *val* is a constant (or a constant list), the translator produces the following AQL statement:

SELECT *attr* FROM *rel* WHERE *attr1* IN (*constlist*),

where *constlist* represents the elements of *val* separated by commas.
If *val* is a simple query, the translation is:

SELECT *attr* FROM *rel* WHERE *attr1* IN (*query*).

In the case *fp* is one of the other comparison functions, and *val* is a constant (or a list), the translation is:

SELECT *attr* FROM *rel* WHERE *attr1 comp* ?,

where *comp* represents the SQL corresponding to the AQL *fp* (< for LT, < = for LE, = for EQ, etc..).
This transaction scheme is applied by the monitor to every element of the list.

If *val* is a query, the same scheme is produced and put in a stack, while the elaboration is resumed on the remaining part of the canonical form.
At the end of the process, the last element of this stack of schemes (where necessarily *val* is a constant list) is executed: the next scheme is applied to its result, and so on.

ii) Queries with more conditions

We suppose, for simplicity, that the query is:

(*attr* OF *rel*) WHEN (*attr1 fp val*) *fl attr2 fp1 val1* ,

where *fl* represents a logical dyadic function (AND, OR).

The translation gives the following schema:

SELECT *attr* FROM *rel* WHERE *attr1 comp* ? *fl attr2 comp1* ?.

Furthermore, the combination table is produced, by executing the cartesian product of *val* and *val1* , structured according to the criteria seen before.

iii) Queries with more attributes

We consider, in this case, the query:

((*attr1* WITH *attr2*) OF *rel*) WHEN *condition_ list.*

The translation in this case gives:

SELECT *attr1,attr2* FROM *rel* WHERE *condition_ list* ,

where, for *condition_ list* , is still valid what we noted before.

In AQL, the function WITH can be used to express "join" operations; in this case, it has as arguments the attribute lists belonging to two different relations, as shown in the following example:

((*attr1* WITH *attr2*) OF *rel1*)WITH((*attr'* WITH *attr"*)OF *rel2*) WHEN *jcondition_ list.*

In this case, *jcondition_ list* is formed by two parts:
the first (obligatory), which represents the join condition. and has the following structure:

(*attr'* OF *rel2*) *fp attr1* OF *rel1* ,

while the second (optional) represents possible additional conditions and has the structure:

fl condition_ list ,

with the usual meaning of the symbols.
The translation gives the following SQL string:

SELECT *rel1.attr1,rel1.attr2,rel2.attr',rel2.attr''* FROM *rel1,rel2* WHERE *jcond.*

The first part of *jcond* corresponds to the first part of *jcondition_ list* and has the structure:

attr1.rel1 comp attr'.rel2 ,

while the translation of the second part undergoes the same process as before.

iv) Use of the functions

In AQL, it is possible to use any APL elementary or defined function at the left of the query, with the meaning that such a function must operate on the result of the query. This query is always constituted in AQL by the variable name containing the selected values or, in the case that more attributes are requested, by the list of the names corresponding to the variables containing the results.

In SQL, it is possible to use only specified functions (COUNT, AVG, SUM, MAX, MIN, other than +, -, *, /) inside the queries: such functions are called "built-in".
To sample the action of the translator in these cases, we consider the following AQL query:

funct_ name query ,

where *query* is any simple query.
In AQL, the functions corresponding to the built-in functions are: HOWMANY, AVERAGE, SUMUP, MAX, MIN, +, -, x, ÷.

If *funct_ name* is one of these keywords, the statement produced will be:

SELECT *builtin* (*attrname*) FROM ...,

where *builtin* is the SQL function corresponding to the one specified in the query, *attrname* is the attribute name to which such a function is applied, the remaining part of the query is translated according to the previous rules.
On the other hand, if *funct_ name* does not correspond to any built-in function, the translator translates the part of the canonical form corresponding to *query* , and lets the result to be processed by the specified *funct_ name.*

3.3.2. Queries with grouping

In AQL, the grouping is specified by means of the following syntax:

(*attr_ list*) GROUP *attr1* BY *attr2* BY.....BY '',

where *attr_ list* is a list of attribute names (qualified or not by the relation), *attr1, attr2,* the "grouping" attributes (i.e. the attributes with respect to which the grouping is performed), GROUP and BY are AQL keywords.
The previous form is the most general way, as it is possible to ask for a grouping of a set of attributes for the attribute *attr1,* inside of this for the attribute *attr2,* etc...

If the grouping is at only one level the previous statement becomes:

(*attr_ list*) GROUP *attr1* BY '';

which, by means of the use of the function GROUPBY, can be rewritten:

(*attr_ list*) GROUPBY *attr1.*

This way of grouping returns groups of the values of the requested attributes, without identifying them with the values of the grouping attribute (*attr1*); if it is required, the function GROUPBYID is used with the same syntax of GROUPBY.

Furthermore, the grouping can be the left argument of a comparison function in a condition, according to the following scheme:

(*attr_ list*) WHEN ([*funct_ name*] *attr* GROUPBY *attr1*) *fp val* ,

with the usual meaning of the symbols.

We will divide both cases according to whether the grouping is in the target list (request list) or in the condition list.

i) Grouping in the target list

As we already said (see 3.1.), in SQL the results of grouping operations do not contain repetitions: this may cause inconsistency if we like to apply whichever function (different from the built-in functions) to the result.
Therefore, in the scheme of each relation, transparently to the user, an attribute (tids) is added, which will be used in the grouping operations if we want to avoid the automatic exclusion of the repetitions.

A simple grouping, formulated in AQL with:

(*attr_ list*) *groupfn attr1*

(where *groupfn* is for GROUPBY or GROUPBYID) is translated in:

SELECT *attr1,attr_ list,tids* FROM *rel*
GROUP BY *attr1, attr_ list,tids.*

From the SQL result, the column corresponding to *tids* is dropped and, if *groupfn* is GROUPBY, even the column referring to *attr1.*

A recursive grouping, like:

(*attr_ list*) GROUP *attr1* BY *attr2* BY ''

is translated into:

SELECT *attr_ list,tids* FROM *rel*
GROUP BY *attr1, attr2, attr_ list,tids* ,

since SQL performs however a recursive grouping considering the positional order of the grouping attributes; *tids* is in last position because it does not affect the grouping.

ii) Grouping in the condition list

An AQL query with a grouping in the condition list has the following structure:

(*attr_ list*) WHEN ([*funct_ name*] *attr* GROUPBY *attr1*) *fp val.*

In the current version, *funct_ name* can be only a built-in function.
This statement has an immediate correspondence with the equivalent SQL using the keyword HAVING:

SELECT *attr_ list* FROM *rel* GROUP BY *attr1,attr.attr_ list*
HAVING *builtin* (*attr*) *comp* ?,

where *builtin* denotes the SQL built-in function corresponding to *funct_ name.*

The result of built-in functions does not show traces of the exclusion of the repetitions of the grouping result; therefore, it does not need the use of the attribute tids in the grouping list.

3.4. Update operations

The syntax of the update in AQL is the following:

varname_ list REPLACE *query* ,

where *query* gives the element (or the elements) of the relation which will be substituted by the value (or values) contained in each of the variables whose names are contained in *varname_ list*.
Such names, in their turn, can be the result of a query.

To describe the action of the translator, we consider the query:

((*attr_ list*) OF *rel*) WHEN *condition_ list*.

The SQL schema corresponding to the previous statement is:

UPDATE *rel* SET *attr1* = ?, *attr2* = ?, ... WHERE *condition_ list*.

where *attr1*, *attr2* , ... are the elements of *attr_ list;* in such a scheme, the question marks will be substituted in an orderly way by the values of each variable whose name is in *varname_ list* during the execution of the updating process.

Conclusions

The main goal of this work was to develop a system that combines the homogeneity with the host language and the user friendliness of AQL with the power, efficiency and security of System R.

The advantages of such a system seem to justify the cost (in efficiency) due to the translator; such a cost however does not seem to be high.

Such a System has been developed in the frame of a project under way at the IBM Scientific Center of Rome, for the development of a territorial data base management system.

References

1. F. Antonacci, et al.: AQL: A Problem-Solving Query Language for Relational Data Bases, IBM Journal of Research and Development, vol. 22, No. 5, September 1978, 541-559.
2. APL Language, GC26-3847-4, IBM Corp..
3. D. D. Chamberlin, et al.: A History and Evaluation of System R. Comm. ACM, vol. 24, No. 10, 632-646
4. E. F. Codd: A Relational Model of Data for Large Shared Data Banks, Comm. ACM, vol. 13, No. 6, June 1970, 377-387.
5. M. M. Zloof: Query by Example, Proc. NCC, AFIPS, Vol. 44, May 1975, 431-438.
6. S. J. P. Todd: The Peterlee Relational Test Vehicle - A System Overview, IBM Systems Journal, vol. 15, No. 4, 1976, 285-308.
7. M. R. Stonebraker, et al.: The Design and Implementation of INGRES, ACM TODS, vol. 1, No. 3, Sept. 1976, 189-222.
8. VM/370 Commands (General User), GX20-1961, IBM Corp..
9. M. Udo, S. Uno: An Experimental Facility for Inter-virtual-machine Communication between APL and non-APL Systems, Proceedings of APL 80 International Conference. Noordwijkerhout, June 1980, G. van der Linden, Ed., 63-70.